DEPARTAMENTO DE INFORMÁTICA E INGENIERÍA DE SISTEMAS

LÍNEA DE INVESTIGACIÓN

IMPLEMENTACIÓN DE REDES DE PETRI CON TIEMPO EN JAVA

TRABAJO DE INVESTIGACIÓN

Director:
José Luís Villarroel Salcedo

Doctorando:
Ramón Piedrafita Moreno

ÍNDICE

JAVA

1.1 INTRODUCCIÓN A JAVA

Java fue creado en los laboratorios de Sun Microsystems Inc. para intentar contrarrestar la incompatibilidad entre plataformas con las que se encontraban los desarrolladores de software. Normalmente el término Java siempre se asocia al lenguaje de programación, pero Java no es sólo un lenguaje de programación. Java es una arquitectura formada por un conjunto de cuatro tecnologías interrelacionadas entre sí que se presenta en la siguiente ilustración.

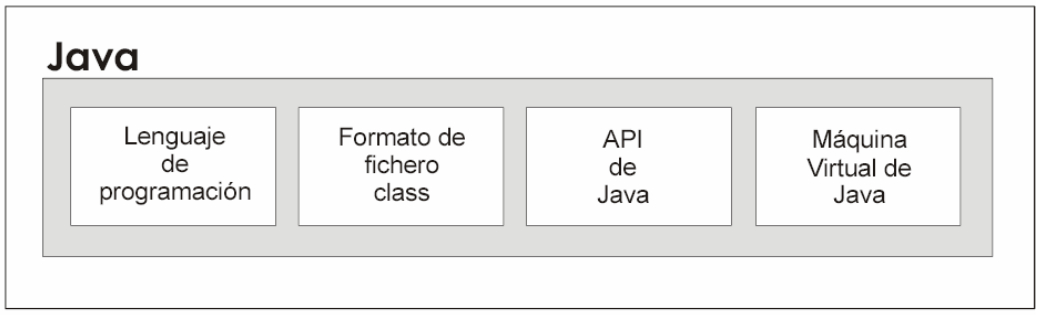

Figura 1 *Tecnologías de Java*

El proceso de desarrollo de un programa en Java se divide en dos etapas:

- compilación: Una vez tenemos el programa codificado en ficheros con extensión java, se procede a compilarlo mediante el compilador de Java. El resultado de la compilación son los ficheros objeto con extensión class que contienen la traducción de Java a bytecode (lenguaje que entiende la Máquina Virtual de Java).

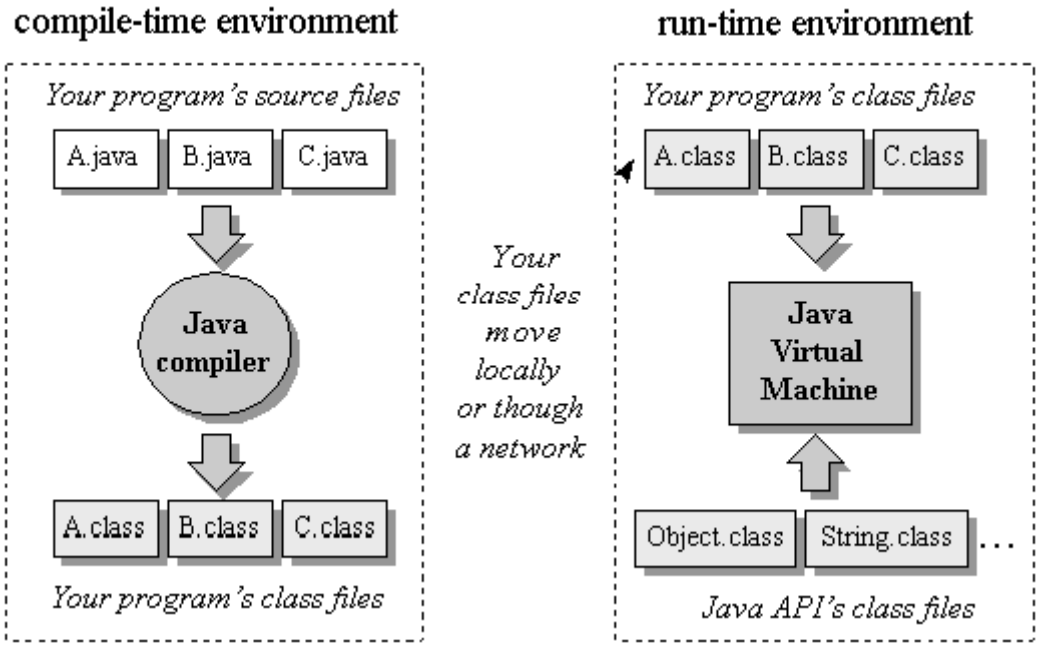

Figura 2 *Entorno de compilación y entorno de ejecución*

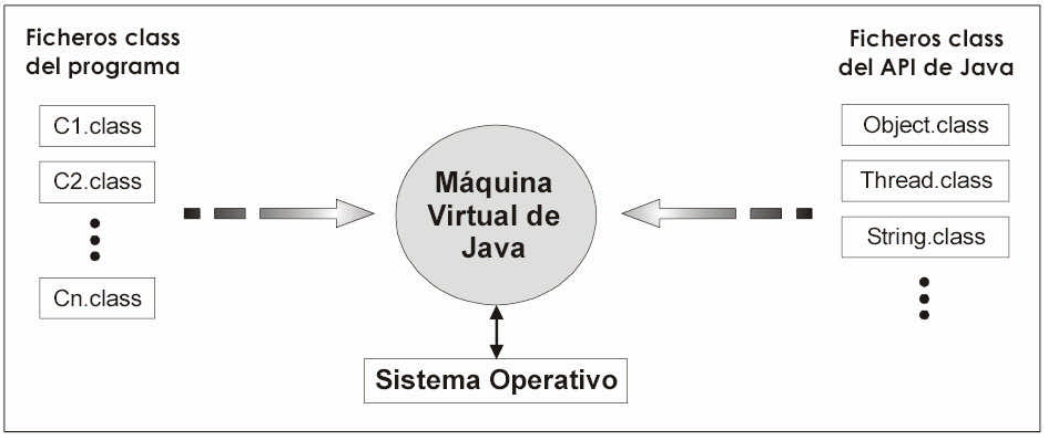

Figura 3 *Ejecución de programas Java*

- Ejecución: Después de la compilación del programa, disponemos de los ficheros class que, junto con los ficheros .class que forman parte del API de Java, serán ejecutados por la *Máquina Virtual de Java* (JVM) (máquina abstracta que ejecuta instrucciones codificadas en bytecode). La *Plataforma Java*, formada por el API de Java junto con la JVM, debe estar presente en cualquier ordenador o dispositivo que quiera ejecutar programas desarrollados en Java.

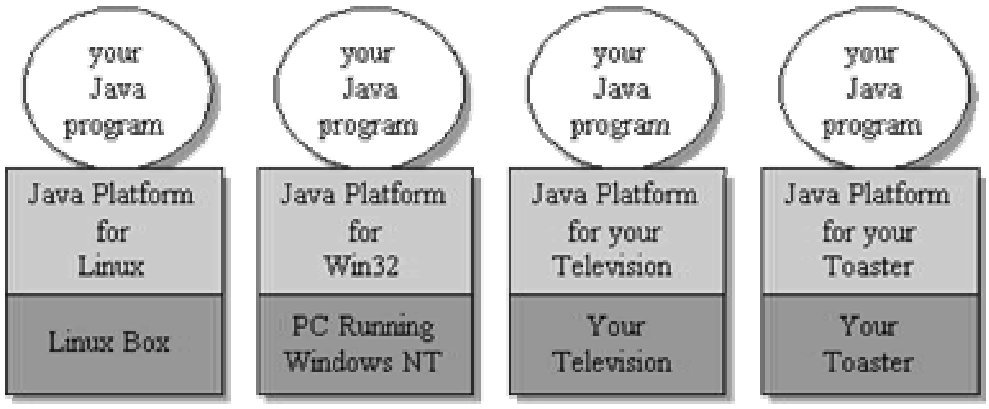

Figura 4 *Plataformas de ejecución*

El lenguaje Java puede pensarse como un lenguaje de propósito general que está orientado para trabajar en red. De tal forma, Java posee algunas características que pueden descartarlo frente a otros lenguajes a la hora de programar una aplicación. A continuación comentaremos algunos de sus inconvenientes:

- El proceso de interpretación de los bytecode produce una ejecución más lenta que la ejecución de la misma aplicación de forma compilada. Con la aparición de la técnica compilación *Just in Time* se acelera la interpretación de bytecode.
- Los programas en Java enlazan las clases dinámicamente, existiendo así la posibilidad de que la JVM tenga que descargar clases remotas retardando el la ejecución de los programas.
- Las verificaciones de los límites de los arrays y referencias a objetos se realizan en tiempo de ejecución. Y no hay forma de eliminarlas lo cual supone incrementar el tiempo de cómputo.
- La gestión de la memoria dinámica se realiza a través del recolector de basura. Se dedica un thread de la JVM a realizar la tarea de recolección de basura, en lugar de ser el programador quien explícitamente se encargue de liberar la memoria ocupada por los objetos. El tiempo de CPU ocupado por el recolector de basura en realizar su función añade cierto grado de incertidumbre e impredecibilidad temporal a la ejecución de un programa multiflujo.

1.2 ESTRUCTURA INTERNA DE LA MÁQUINA VIRTUAL JAVA

La siguiente ilustración muestra la estructura interna de la JVM:

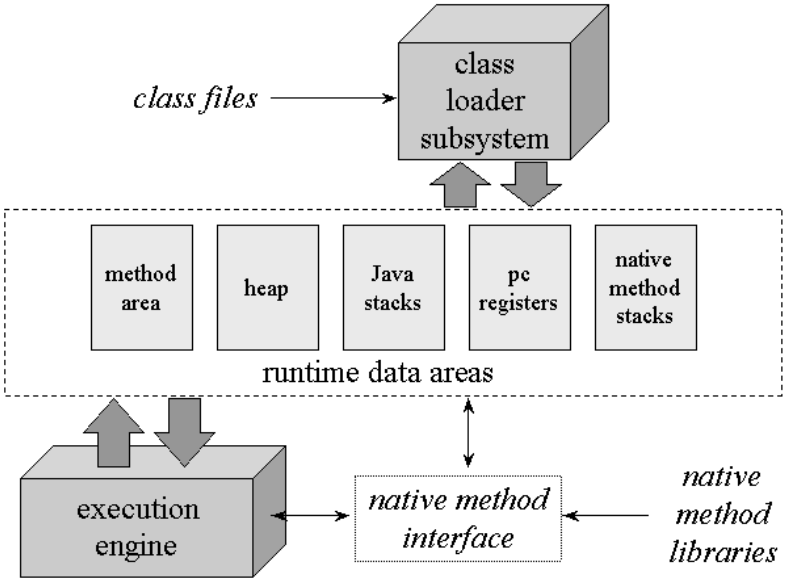

Figura 5 Estructura interna de la Máquina virtual Java

- **Class Loader Subsystem** Sistema encargado de cargar las clases del programa de usuario, del API de Java, sean ambas locales o remotas
- **Execution Engine**: Ejecuta las instrucciones de los métodos de las clases cargadas en la JVM
- **Runtime Data Areas** Áreas de datos en tiempo de ejecución

Una instancia de la JVM tiene una *Method Area* y un *Heap*. Estas zonas de memoria son compartidas por todos los threads que se ejecutan en la JVM. Cuando la máquina virtual carga un

fichero class, extrae información sobre el tipo y la almacena en la Method Area. A medida que la ejecución del programa evoluciona, la JVM carga todas las instancias de los objetos dentro del Heap.

Cada thread nuevo que se cree, dispondrá de su propio *pc register* (registro para el contador de programa) y de una *Java stack* (pila de Java). Cuando un thread esté ejecutando un método no nativo, el valor del registro contendrá una referencia a la siguiente instrucción a ejecutar. La Java stack almacenará el estado de las invocaciones de métodos no nativos de Java que el thread vaya haciendo durante su ejecución. Este estado incluye a las variables locales, los parámetros de los métodos, el valor de retorno (en caso de haberlo) y cálculos intermedios. El estado de invocación de métodos nativos se guarda usando unas estructuras de datos llamadas *native method stacks*.

La *Java stack* está formada por *stack frames*. Una *stack frame* guarda el estado de una invocación de un método. Cuando un thread invoca a un método, la JVM empila una nueva stack frame en la Java stack de ese thread y, cuando finalice la ejecución de dicho método, la JVM desempilará dicha stack frame de la Java stack.

La JVM no dispone de registros para almacenar cálculos intermedios, con lo que el conjunto de instrucciones de la JVM usa la *Java stack* para almacenar esos posibles cálculos intermedios que genere la ejecución de cada thread.

Una instancia de la JVM empieza a ejecutar la aplicación Java invocando el método main:

public static void main(String[] args)

es el punto de comienzo de una aplicación Java. El método main estará asignado al thread inicial de la aplicación, que será el encargado de iniciar el resto de threads de la aplicación.

Hay dos tipos de threads dentro de la máquina virtual: ***daemon*** y ***non-daemon***. Un ejemplo de thread daemon que es usado por la maquina virtual Java, es el encargado de la función de ***garbage collection*** (proceso automático que se encarga de liberar el espacio ocupado por los objetos que ya no son referenciados dentro de la aplicación Java). De todas formas, la aplicación Java puede marcar un thread como daemon (el thread inicial de una aplicación es un thread del tipo non-daemon). La instancia de la máquina virtual continúa activa mientras exista algún thread non-daemon que se esté ejecutando. Cuando todos los threads non-daemon de la aplicación terminen, entonces la instancia de la máquina virtual de java finalizará su ejecución.

1.3 PROCESOS E HILOS

Un proceso es un programa ejecutándose dentro de su propio espacio de direcciones. Java es un sistema multiproceso, esto significa que soporta varios procesos corriendo a la vez dentro de sus propios espacios de direcciones.

La multitarea es soportada en Java mediante el concepto de hilo. Un hilo es un único flujo de ejecución dentro de un proceso.

Un hilo es una secuencia de código en ejecución dentro del contexto de un proceso (programa). Los hilos no pueden ejecutarse ellos solos, requieren la supervisión de un proceso padre para correr. Dentro de cada proceso puede haber varios hilos ejecutándose. Los hilos dependen de un proceso padre para acceder a los recursos de ejecución.

Normalmente los hilos son implementados a nivel de sistema, necesitando una interfaz de programación específica del sistema separada del lenguaje de programación. Java se presenta como ambos, como lenguaje y como sistema de tiempo de ejecución. Java es un lenguaje de programación que incorpora hilos en el mismo lenguaje.

1.3.1 THREAD

Dentro del API de Java se define la clase Thread mediante la cual es posible definir flujos de ejecución. Para añadir la funcionalidad de hilo a una clase simplemente se deriva la clase de Thread y

se incorpora el método run. Es en este método run donde el procesamiento de un hilo toma lugar, y a menudo se refieren a él como el cuerpo del hilo.

Control de Threads

- Creación de un thread (constructor Thread).
- Lanzar la ejecución (start).
- Detener la ejecución (suspend, interrupt, stop, sleep). (suspend y stop han sido desechados)
- Reanudar la ejecución (resume).
- Ceder el procesador a otros hilos (yield).
- Esperar la finalización de otro thread (join).

Gestión y consulta de propiedades:

- Nombre (getName / setName).
- Prioridad (getPriority / setPriority): En java la prioridad mínima es 1 (MIN_PRIORITY), la normal es 5 (asignada por defecto) (NORM_PRIORITY) y la prioridad máxima es 10 (MAX_PRIORITY)
- Daemon: isDaemon indica si el thread es de sistema o no; setDaemon convierte a un thead en daemon

1.3.2 CREACIÓN DE HILOS

En Java existen dos sistemas para llevar a cabo la creación de un hilo:

Extensión la clase Thread.

Implementando en una clase la interfaz Runnable

Extensión de la clase Thread

Se crea una clase que extiende la clase Thread Por ejemplo,

```
class Mihilo extends Thread {
public void run() {
. . .
}
}
```

Es la declaración clase, *Mihilo*, que extiende la clase Thread. Se deberá sobrescribir el método *run* heredado de Thread. Es en este método *run* donde se implementa el código correspondiente a la acción (la tarea) que el hilo debe desarrollar. El método run no debe ser invocado directamente. La ejecución de un hilo se lanza con el método Star.

En el caso de crear un hilo extendiendo la clase Thread, se heredan los métodos y variables de la clase padre (thread). Esta subclase solamente puede extender o derivar una vez de la clase padre Thread.

Implementación de la interfaz runnable

Las interfaces representan una forma de encapsulamiento del trabajo que una clase debe realizar. Así, se utilizan para el diseño de requisitos comunes a todas las clases que se tiene previsto implementar. La interfaz define el trabajo, la funcionalidad que debe cubrirse, mientras que la clase o clases que implementan la interfaz realizan dicho trabajo (cumplen esa funcionalidad). Todas las clases o grupos de clases que implementen una cierta interfaz deberán seguir las mismas reglas de funcionamiento.

```
public class Mihilo implements Runnable {
Thread t;
public void run() {
// Ejecución del thread una vez creado
}
}
```

En este caso necesitamos crear una instancia de Thread antes de que el sistema pueda ejecutar el proceso como un hilo. El método run tendrá que implementarse en la nueva clase creada. Y también el método Star que encapsulará la llamada al método Star de la clase Thread

La diferencia entre ambos métodos de creación de hilos en Java radica en la mayor flexibilidad de que dispone el programador en el caso de la utilización de la interfaz Runnable. Es una buena praxis que las clases que necesiten ejecutarse como un hilo implementen la interfaz Runnable, ya que de esta forma se permite que sean extendidas por subclases.

Las interfaces no realizan ninguna tarea en la ejecución del programa. Las interfaces proporcionan una idea de diseño, de infraestructura, de la clase Thread, pero ninguna funcionalidad.

Una vez definida la clase del hilo en el main del programa (o en otros hilos) se deben crear las instancias necesarias del hilo, por ejemplo:

procesoCode proceso1= new procesoCode("proceso Code uno",com1, buffer);

Llama al constructor y crea un hilo de la clase procesoCode

1.3.3 CONTROL DE UN HILO

Arranque de un hilo

El procedimiento Star arranca la ejecución del hilo en el scheduler de java

Proceso1.start();

Y además llama a la ejecución del método run.

Si se llama directamente al método run de un hilo, no se activa la ejecución del hilo, es como llamar a un método de un objeto. Las tareas a realizar por el hilo deben estar programadas en el método run. Cuando finalice run, finalizará también el hilo que lo ejecutaba.

Suspensión de un Hilo

La función miembro suspend de la clase Thread permite tener un control sobre el hilo de modo que podamos desactivarlo, detener su actividad durante un intervalo de tiempo indeterminado, El hilo es suspendido indefinidamente y para volver a activarlo de nuevo necesitamos realizar una invocación a la función miembro resume.

Una llamada al método sleep, lleva al hilo a un estado de “dormido” durante un tiempo determinado.

Parada de un Hilo

El método stop debería terminar la ejecución de un hilo de forma asíncrona:

Proceso1.stop();

Este método ha sido desechado en las últimas versiones de Java.

Alive

Proceso1.isAlive();

Devolverá true en caso de que el hilo *Proceso1*esté vivo

Prioridad de Hilos y Planificación

Cada hilo tiene una prioridad, un valor entero entre 1 y 10, de modo que cuanto mayor el valor, mayor es la prioridad. Cuando se crea un hilo en Java, éste hereda la prioridad de su padre, el hilo que lo ha creado. A partir de aquí se le puede modificar su prioridad en cualquier momento utilizando el método setPriority **.** Las prioridades de un hilo varían en un rango de enteros comprendido entre MIN_PRIORITY y MAX_PRIORITY (ambas definidas en la clase Thread). El entero más alto designará la prioridad más alta y el más bajo, como es de esperar, la menor.

El Planificador de Java (*Scheduler)*, decide que hilos deben ejecutarse dentro del conjunto de los que se encuentran preparados para su ejecución. La regla básica del planificador es que si solamente hay hilos daemon ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. El planificador determina qué hilos deberán ejecutarse comprobando la prioridad de todos los hilos que estén preparados para ejecutarse. Aquellos con prioridad más alta dispondrán del procesador antes de los que tienen prioridad más baja.

Se ejecutará primero el hilo de prioridad superior, el llamado “Ejecutable”, y sólo cuando éste para, abandona o se convierte en “No Ejecutable”, comienza la ejecución de los hilos de prioridad inferior. Si dos hilos tienen la misma prioridad, el programador elige uno de ellos en alguna forma de competición. El hilo seleccionado se ejecutará hasta que:

Un hilo con prioridad mayor pase a ser “Ejecutable”.

En sistemas que soportan tiempo-compartido, termina su tiempo.

Abandone, o termine su método run.

El planificador puede seguir dos patrones, ***preemptivo (expulsivo)*** ó ***no preemptivos***. Los planificadores no preemptivos proporcionan un segmento de tiempo a todos los hilos que están corriendo en el sistema. El planificador decide cuál será el siguiente hilo a ejecutarse y llama a resume para darle vida durante un período fijo de tiempo. Cuando finaliza ese período de tiempo, se llama a su método suspend y el siguiente hilo en la lista de procesos será relanzado mediante su método resume. Los planificadores preemptivos, en cambio, deciden qué hilo debe correr según su prioridad y lo ejecutan hasta que concluye. El hilo tiene control total sobre el sistema mientras esté en ejecución. El método yield es un mecanismo que permite a un hilo forzar al planificador para que comience la ejecución de otro hilo que esté esperando. Dependiendo del sistema en que esté corriendo Java, el planificador será ***preemptivo*** o no ***preemptivo***.

SINCRONIZACIÓN

2.1 INTRODUCCIÓN

El problema de la sincronización de hilos tiene lugar cuando varios hilos intentan acceder al mismo recurso o dato. A la hora de acceder a datos comunes, los hilos necesitan establecer cierto orden, por ejemplo en el caso del productor consumidor. Para asegurarse de que hilos concurrentes no se estorban y operan correctamente con datos (o recursos) compartidos, un sistema estable previene la inanición y el punto muerto o interbloqueo. La inanición tiene lugar cuando uno o más hilos están bloqueados al intentar conseguir acceso a un recurso compartido de ocurrencias limitadas. El interbloqueo es la última fase de la inanición; ocurre cuando uno o más hilos están esperando una condición que no puede ser satisfecha. Esto ocurre muy frecuentemente cuando dos o más hilos están esperando a que el otro u otros se desbloquee, respectivamente.

2.2 PROGRAMACIÓN

El siguiente programa esta compuesto de varios Threads. Un thread coordinador autoriza la ejecución del código de otros dos threads. Al principio de la ejecución del programa se crean los threads:

```
//Se crea el coordinador
coordinador coor=new coordinador("COORDINADOR ",com1,com2, buffer);
//Creación de los procesos code
procesoCode proceso1= new procesoCode("proceso Code uno",com1, buffer);
procesoCode proceso2= new procesoCode("proceso Code dos",com2, buffer);
//Se les asignan prioridades de ejecución
proceso1.setPriority(5);
proceso2.setPriority(2);
```

```
    // asignamos al coordinador la máxima prioridad
    coor.setPriority(10);
    // Se lanza la ejecución de los theads
    proceso2.start();
    proceso1.start();
    coor.start();


  }
```

La creación de los threads supone una llamada a su método constructor. En el caso del coordinador:

```
  class coordinador extends Thread{
// la clase coordinador deriva de la clase Thread
  // variables privadas del coordinador
    private String nombre;
    private int valor;
    private int i,j;
    private comunica c1;
        private comunica c2;
    private BoundedBuffer buffercoor;
  //método constructor
  coordinador(String anombre,comunica ac1, comunica ac2, BoundedBuffer abuffercoor){
        nombre=anombre;
        c1=ac1;
        c2=ac2;
        buffercoor=abuffercoor;
    }
```

Se crea un hilo que tiene las variables privadas nombre, c1,c2, y buffercoor

En el caso de los procesos code:

```
  class procesoCode extends Thread{

    private String nombre;
    private comunica c;

    private BoundedBuffer bufferproceso;
    private int valor;private int i;
  //método constructor
    procesoCode(String anombre,comunica ac,BoundedBuffer abuffer){
        nombre=anombre;
        c=ac;
```

```
        bufferproceso=abuffer;
    }
```

Se crea dos hilos que cada uno tiene sus variables privadas nombre, c, y buffercoor

Al lanzar la ejecución de los hilos mediante su método star

```
proceso2.start();
proceso1.start();
coor.start();
```

Se ejecutan los tres hilos en paralelo. El sistema de ejecución de nuestro Java asigna un porcentaje de CPU a todos los hilos. En principio parece que este porcentaje es proporcional a la prioridad de la CPU. No una planificación expulsiva, por lo tanto aunque el coordinador tenga mayor prioridad, el scheduler (planificador) ejecuta los tres hilos.

Para posibilitar la comunicación entre los diferentes hilos se utilizan dos tipos de objetos:

Monitores

Buffer

2.2.1 IMPLEMENTACIÓN DEL MONITOR

```
    class comunica {
// monitores utilizados por el coordinador para autorizar la ejecución
    // de procesos code
      private
 int dato =1;
    private boolean disponible = false;
    public synchronized void preguntautorizacion()
    {
        while(!disponible){
                try
                {
                        wait();
                }
                catch(InterruptedException e){}
        }
            disponible=false;
            notify();
    }

    public synchronized void autorizoejecucion()
    {
            while(disponible){
                try                             {
                        wait();
                }
                catch(InterruptedException e){}
            }
                disponible=true;
                notify();
    }
```

}

Un monitor no es un Thread, es un objeto pasivo, no activo. El monitor *comunica* dispone de dos métodos sincronizados que impiden que varios hilos accedan simultáneamente a los métodos sincronizados.

La declaración de estos métodos se realiza colocando la palabra *syncrhonized* antes del tipo de dato devuelto por el método.

Cuando en una clase hay varios métodos synchronized se asegura que no van a ejecutarse simultáneamente (por varios threads) más de 1 de ellos.

También es posible definir fragmentos de código sincronizados con

```
synchronized (expr) {
sentencias }
```

Cuando el coordinador autoriza la ejecución de un proceso llama a autorizoejecucion().

```
System.out.println (nombre + " dice al proceso 1 que comience") ;
c1.autorizoejecucion();
System.out.println (nombre + " dice al proceso 2 que comience") ;
c2.autorizoejecucion();
```

La variable disponible se pone a true y notify despierta a los threads que estén esperando. Estos threads son los procesos code que están esperando que el coordinador los autorice:

```
public void run(){
while(true){      System.out.println (nombre + " esperando autorización") ;
        c.preguntautorizacion();
```

Cuando un proceso code al principio de cada bucle de su método *run()* ejecuta:

```
c.preguntautorizacion();
```

En el caso de que la variable disponible sea false entonces se ejecuta:

```
try
{
        wait();
}
catch(InterruptedException e){}
```

Try indica que es un bloque de código donde se prevé que se genere una excepción. El bloque *try* tiene que ir seguido, al menos, por una cláusula *catch* o una cláusula *finally.*

Catch es el código que se ejecuta cuando se produce la excepción. No puede existir código entre el final del bloque *try* y el bloque *catch*, ni entre bloques *catch.*

wait

La ejecución de *wait* supone:

Bloqueo inmediato del thread que lo ejecuta

Si el thread actual (que ejecuta el wait) es interrumpido entonces el thread abandona inmediatamente el wait y se genera la excepción InterruptedException.

La máquina virtual Java pone el thread en el conjunto de espera interno, haciéndolo inaccesible.

El objeto monitor libera el cerrojo de sincronización.

Notify

Una invocación a notify() da lugar a:

Si existe algún hilo en el conjunto de espera interno asociado al objeto cerrojo, la JVM elimina del conjunto de espera un thread arbitrario. No hay ninguna forma de saber que thread será liberado en el caso de que haya varios en espera.

El thread liberado obtendrá el cerrojo de sincronización una vez que el thread que llamo a notify los libere. El thread que ejecuta el notify tendrá que salir del procesador para que el thread liberado lo ocupe y obtenga el cerrojo. Hasta entonces se bloqueará. Pero puede seguir bloqueado si algún otro thread obtiene primero el cerrojo, por ejemplo si otro thread ocupa el procesador antes.

Una vez obtenido el cerrojo el thread continua su actividad en el punto de programa siguiente a su wait()

NotifyAll

Funciona de forma idéntica a notify solo que simultáneamente en todos los threads que estén en espera, pero solo uno de ellos, si ocupa el procesador a continuación del thread que invoca a notifyall, obtendrá el cerrojo del monitor.

Esperas temporizadas

El método wait admite argumentos temporales de forma que un thread permanezca en el conjunto de espera un máximo tiempo.

wait(long milisegundos)

wait(long milisegundos, long nanosegundos)

Un thread en espera temporizada puede reanudar su ejecución un tiempo arbitrario después del plazo especificado, como consecuencia de competencias entre threads por el procesador, política de planificación aplicada y granuralidad de la temporización

2.2.2 BUFFER

Cuando un proceso code termina avisa al coordinador que ha terminado su ejecución almacenando su identidad en un buffer:

```
class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;
```

Todas las variables son privadas excepto el tamaño del buffer para que pueda ser consultado por el coordinador. A continuación tenemos el constructor del buffer:

```
  public BoundedBuffer(int length) {
    size = length;
```

```
    buffer = new int[size];
    last = 0;
    first = 0;
  }
```

El método sincronizado *put* es utilizado por los procesos code para almacenar su identidad. Código en los procesos code:

```
                    if (nombre=="proceso Code dos"){
                    bufferproceso.put(2);}
                    if (nombre=="proceso Code uno"){
                    bufferproceso.put(1);}


public synchronized void put(int item)
 {
   while(numberInBuffer == size){
                    try
                    {
                        wait();
                    }
                    catch(InterruptedException e){}
               }
   last = (last + 1) % size ; // % modulo
   numberInBuffer++;
   buffer[last] = item;
   notify();
 }
```

El método sincronizado *get* es utilizado por el coordinador para saber que procesos code han terminado. Código en el coordinador:

```
        while(buffercoor.numberInBuffer>0) {
                    valor= buffercoor.get();
            if (valor==2){
            System.out.println (nombre + " el proceso 2 ha finalizado") ;
            }
            if (valor==1){
            System.out.println (nombre + " el proceso 1 ha finalizado") ;
            }
            }
```

```
public synchronized int get()
{
   while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
   first = (first + 1) % size ; // % modulo
   numberInBuffer--;
   notify();
   return buffer[first];
 }
}
```

2.3 LISTADO DE PROGRAMA

```
class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }
     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }
     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}
class comunica {
// monitores utilizados por el coordinador para autorizar la ejecución
// de procesos code
 private
  int dato =1;
private boolean disponible = false;
public synchronized void preguntautorizacion()
        {
          while(!disponible){
                        try
                        {
```

```
                        wait();
                    }
                    catch(InterruptedException e){}
                }
                disponible=false;
                notify();
        }
public synchronized void autorizoejecucion()
        {
                while(disponible){
                    try
                    {
                        wait();
                    }
                    catch(InterruptedException e){}
                }
                    disponible=true;
                    notify();
        }
}

class coordinador extends Thread{

private String nombre;
private int valor;
private int i,j;
private comunica c1;
private comunica c2;
private BoundedBuffer buffercoor;
coordinador(String anombre,comunica ac1, comunica ac2, BoundedBuffer abuffercoor){
        nombre=anombre;
        c1=ac1;
        c2=ac2;
        buffercoor=abuffercoor;
        }

public void run(){
        System.out.println (nombre + " empieza ejecución");
        for (j = 0; j < 2; j++){//el coordinador se ejecuta dos veces

        System.out.println (nombre + " dice al proceso 1 que comience") ;
        c1.autorizoejecucion();
        System.out.println (nombre + " dice al proceso 2 que comience") ;
        c2.autorizoejecucion();

            for (i = 0; i < 100000; i++) {
            // esperamos un rato
                }
        while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
            if (valor==2){
            System.out.println (nombre + " el proceso 2 ha finalizado") ;
                }
            if (valor==1){
            System.out.println (nombre + " el proceso 1 ha finalizado") ;
                }
            }
            }
        }

}// end coordinador
```

```
class procesoCode extends Thread{
private String nombre;
private comunica c;
private BoundedBuffer bufferproceso;
private int valor;private int i;
procesoCode(String anombre,comunica ac,BoundedBuffer abuffer){
        nombre=anombre;
        c=ac;
        bufferproceso=abuffer;
       }
public void run(){
while(true){System.out.println (nombre + " esperando autorización") ;
        c.preguntautorizacion();
           System.out.println (nombre + " ejecutando codigo") ;
              for (i = 0; i < 2; i++) {
               System.out.println (i + "ejecutando codigo" + nombre) ;
              }

        System.out.println (nombre + " finalizado") ;
        if (nombre=="proceso Code dos"){
        bufferproceso.put(2);}
        if (nombre=="proceso Code uno"){
        bufferproceso.put(1);}
       }
       }
}

class sincro{
public static void main (String args[]){
comunica com1=new comunica();
comunica com2=new comunica();
BoundedBuffer buffer= new BoundedBuffer(2);
coordinador coor=new coordinador("COORDINADOR ",com1,com2, buffer);
procesoCode proceso1= new procesoCode("proceso Code uno",com1, buffer);
procesoCode proceso2= new procesoCode("proceso Code dos",com2, buffer);
proceso1.setPriority(5);
proceso2.setPriority(2);
proceso2.start();
proceso1.start();
coor.setPriority(1);
coor.start();
}
}
```

Ejemplo de ejecución:

```
ramonpor:~/javahecho2/ java sincro
proceso Code dos esperando autorización
proceso Code uno esperando autorización
COORDINADOR  empieza ejecución
COORDINADOR  dice al proceso 1 que comience
COORDINADOR  dice al proceso 2 que comience
proceso Code dos ejecutando codigo
0ejecutando codigoproceso Code dos
1ejecutando codigoproceso Code dos
proceso Code dos finalizado
proceso Code dos esperando autorización
```

COORDINADOR el proceso 2 ha finalizado
COORDINADOR empieza ejecución
COORDINADOR dice al proceso 1 que comience
proceso Code uno ejecutando codigo
0ejecutando codigoproceso Code uno
1ejecutando codigoproceso Code uno
proceso Code uno finalizado
proceso Code uno esperando autorización
COORDINADOR dice al proceso 2 que comience
proceso Code dos ejecutando codigo
0ejecutando codigoproceso Code dos
1ejecutando codigoproceso Code dos
proceso Code dos finalizado
proceso Code dos esperando autorización
COORDINADOR el proceso 1 ha finalizado
COORDINADOR el proceso 2 ha finalizado
proceso Code uno ejecutando codigo
0ejecutando codigoproceso Code uno
1ejecutando codigoproceso Code uno
proceso Code uno finalizado
proceso Code uno esperando autorización

IMPLEMENTACIÓN CENTRALIZADA

3.1 PROBLEMA DE LOS FILÓSOFOS RESUELTO POR EL MÉTODO DE LAS GUARDAS

Las transiciones de la red de Petri representan que un filósofo esta comiendo o esta pensando. Están implementadas mediante threads de Java. La coordinación del disparo de las transiciones se realiza mediante un coordinador implementado también en un thread de Java.

La comunicación entre el thread coordinador y los threads transición se realiza mediante monitores y un buffer. El coordinador examina en el test de habilitación que transiciones se pueden disparar. Cuando el coordinador decide que una transición debe dispararse, desmarca sus etapas de entrada y se lo comunica al thread transición mediante un monitor. Entonces la transición ejecuta su código y cuando finaliza la ejecución se lo comunica al coordinador escribiendo su identidad en un buffer.

A continuación del test de habilitación el coordinador comprueba si en el buffer se ha almacenado algún número lo cual indica que hay transiciones que han finalizado su ejecución. Entonces el coordinador actualiza el estado de la red de Petri marcando sus etapas de salida.

```
class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
```

```
   buffer = new int[size];
   last = 0;
   first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }
     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }

     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}

class comunica {
// monitores utilizados por el coordinador para autorizar la ejecución
// de transciones
private boolean disponible = false;
public synchronized void preguntautorizacion()
        {
          while(!disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
                disponible=false;
                notify();

        }
public synchronized void autorizoejecucion()
        {
                while(disponible){
```

```
                    try
                    {
                            wait();
                    }
                    catch(InterruptedException e){}
                }

                    disponible=true;
                    notify();
        }
}

class transicion extends Thread{
private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

transicion(int aidentidad,comunica ac,BoundedBuffer abuffer){
        identidad=aidentidad;
        bufferproceso = abuffer;
        c =ac;
       }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){//System.out.println (identidad + " esperando autorización") ;
        c.preguntautorizacion();
switch(identidad){
            case TF1_P:System.out.println ("filósofo uno pensando") ;break;
            case TF2_P:System.out.println ("filósofo dos pensando") ;break;
            case TF3_P:System.out.println ("filósofo tres pensando") ;break;
            case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
            case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
            case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
            case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
            case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
            case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
            case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
                }

            for (i = 0; i < 100000000; i++) {
            // comiendo o pensando
                }
        bufferproceso.put(identidad);
       }
       }
}

class coordinador extends Thread{
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];

private BoundedBuffer buffercoor;
```

```
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
        nombre=anombre;
        vectorcomunica=avectorcomunica;
        M=aestado;
        buffercoor=abuffercoor;
       }

public void run(){
System.out.println (nombre + " empieza ejecución");

while(true){

            if (M[P3]) {
            System.out.println (  "filósofo tres autorizado a pensar") ;

            vectorcomunica[TF3_P].autorizoejecucion();M[P3]= false;
                }

            if (M[P1]) {
            System.out.println (  "filósofo uno autorizado a pensar") ;

            vectorcomunica[TF1_P].autorizoejecucion();M[P1]= false;
                }

            if (M[P2]) {
            System.out.println (  "filósofo dos autorizado a pensar") ;

            vectorcomunica[TF2_P].autorizoejecucion();M[P2]= false;
                }

            if (M[P4]) {
            System.out.println (  "filósofo cuatro autorizado a pensar") ;

            vectorcomunica[TF4_P].autorizoejecucion();M[P4]= false;
                }

            if (M[P5]) {
            System.out.println (  "filósofo cinco autorizado a pensar") ;

            vectorcomunica[TF5_P].autorizoejecucion();M[P5]= false;
                }
               if    (M[T2] && M[T3] && M[C3]){
            System.out.println (  "filósofo tres autorizado a comer") ;

            vectorcomunica[TF3_C].autorizoejecucion();M[T2]= false;M[T3]= false;M[C3]= false;
                }

               if    (M[T5] && M[T1] && M[C1]){
            System.out.println (  "filósofo uno autorizado a comer") ;

            vectorcomunica[TF1_C].autorizoejecucion();M[T5]= false;M[T1]= false;M[C1]= false;
                }

               if    (M[T1] && M[T2] && M[C2]){
            System.out.println (  "filósofo dos autorizado a comer") ;

            vectorcomunica[TF2_C].autorizoejecucion();M[T1]= false;M[T2]= false;M[C2]= false;
                }
```

```
                if    (M[T3] && M[T4] && M[C4]){
            System.out.println (  "filósofo cuatro autorizado a comer") ;

            vectorcomunica[TF4_C].autorizoejecucion();M[T3]= false;M[T4]= false;M[C4]= false;
                }

                if    (M[T4] && M[T5] && M[C5]){
            System.out.println (  "filósofo cinco autorizado a comer") ;

            vectorcomunica[TF5_C].autorizoejecucion();M[T4]= false;M[T5]= false;M[C5]= false;
                }

            if (buffercoor.numberInBuffer>0) System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );

// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????
while(buffercoor.numberInBuffer>0) {
                valor= buffercoor.get();
          switch(valor){
                case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
                case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
                case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
                case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
                case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
                case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
                case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
                case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
                case TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println (  "filósofo cuatro termina de comer");break;
                case TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println (  "filósofo cinco termina de comer");break;
                }// end swicth


            }//end while
            }//end while true
          } //en run

}// end coordinador


class guardas5{


public static void main (String args[]){
    int i;

// definición de contantes para las transiciones
int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

// definición de contantes para los lugares
int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;

// estado incial de la red de petri
boolean[] estado={true,false,true,true,false,true,true,false,true,true,false,true,true,false,true};

// vector de monitores para comunicar la acepatación del comienzo del disparo de las transiciones


comunica com1=new comunica();
comunica com2=new comunica();
comunica com1p=new comunica();
comunica com1c=new comunica();
comunica com2p=new comunica();
```

```
comunica com2c=new comunica();
comunica com3p=new comunica();
comunica com3c=new comunica();
comunica com4p=new comunica();
comunica com4c=new comunica();
comunica com5p=new comunica();
comunica com5c=new comunica();

comunica[] comunicaarray = {com1p,com1c,com2p,com2c,com3p,com3c,com4p,com4c,com5p,com5c};



// buffer para comunicar los fin de code
BoundedBuffer buffer= new BoundedBuffer(10);



transicion TF1_Ptask= new transicion(TF1_P,comunicaarray[0], buffer);
transicion TF1_Ctask= new transicion(TF1_C,comunicaarray[1], buffer);
transicion TF2_Ptask= new transicion(TF2_P,comunicaarray[2], buffer);
transicion TF2_Ctask= new transicion(TF2_C,comunicaarray[3], buffer);
transicion TF3_Ptask= new transicion(TF3_P,comunicaarray[4], buffer);
transicion TF3_Ctask= new transicion(TF3_C,comunicaarray[5], buffer);
transicion TF4_Ptask= new transicion(TF4_P,comunicaarray[6], buffer);
transicion TF4_Ctask= new transicion(TF4_C,comunicaarray[7], buffer);
transicion TF5_Ptask= new transicion(TF5_P,comunicaarray[8], buffer);
transicion TF5_Ctask= new transicion(TF5_C,comunicaarray[9], buffer);

transicion[]                                                                    Transicionarray=
{TF1_Ptask,TF1_Ctask,TF2_Ptask,TF2_Ctask,TF3_Ptask,TF3_Ctask,TF4_Ptask,TF4_Ctask,TF5_Ptask,TF5_Ctask};
coordinador coor=new coordinador("COORDINADOR FILÓSOFOS",comunicaarray,estado,buffer);
coor.setPriority(10);



for (i = 0; i < 10; i++) {
                Transicionarray[i].setPriority(5);
                Transicionarray[i].start();
               }
coor.start();



}
}
```

Ejemplo de ejecución:

```
ramonpor:~/javahecho2/ javac guardas4.java
ramonpor:~/javahecho2/ java guardas4
transición 1 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 3 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
COORDINADOR FILÓSOFOS empieza ejecución
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo cinco autorizado a pensar
filósofo cinco pensando
final de 2 transiciones
false false true false false true false false true false false true false false true
filósofo uno termina de pensar
filósofo cinco termina de pensar
filósofo uno autorizado a comer
filósofo uno comiendo
final de 3 transiciones
false false false false false true false false true false false true false true false
filósofo tres termina de pensar
filósofo dos termina de pensar
filósofo cuatro termina de pensar
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false true false false false false false true true false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false true false false false false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo dos autorizado a comer
filósofo dos comiendo
final de 2 transiciones
```

false false false false false false false false true false true false false false false
filósofo uno termina de pensar
filósofo cinco termina de comer
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 2 transiciones
false true false false false false false false false false false false false false true
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo uno autorizado a comer
filósofo uno comiendo
final de 1 transiciones
false false false false false true false true false false false false false false false
filósofo cinco termina de pensar
final de 2 transiciones
false false false false false true false true false false false false false true false
filósofo dos termina de pensar
filósofo cuatro termina de comer
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false true false false false false false false true false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false false false false false false
filósofo cuatro termina de pensar
final de 1 transiciones
false false true false true false false false false false true false false false false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo dos autorizado a comer
filósofo dos comiendo
final de 2 transiciones
false false false false false false false false true false true false false false false
filósofo uno termina de pensar
filósofo cinco termina de comer
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 1 transiciones

false true false false false false false false false false false false false true
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo uno autorizado a comer
filósofo uno comiendo
final de 2 transiciones
false false false false false true false false false false false false false false
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false false false false false false false true false false false
filósofo cinco termina de pensar

Examinamos el test de habilitación de las transiciones

```
if (M[P3]) {
        System.out.println (  "filósofo tres autorizado a pensar") ;

        vectorcomunica[TF3_P].autorizoejecucion();M[P3]= false;
        }
```

Se testea el marcado de las etapas de entrada, se autoriza la ejecución de la transición y se desmarca las etapas de entrada. Pero cuando se autoriza la ejecución de la transición, esta toma el procesador y la red de Petri puede no llegar a desmarcarse antes de que la transición finalice. Recordemos que en java clásico la prioridad de los threads no significa prioridad de ejecución sino "más porcentaje de tiempo de CPU". No se da el caso que esten marcadas las etapas de entrada y de salida de una transición dado que cuando el coordinador retome la CPU empezará por desmarcar la etapa de entrada. Tampoco habrá problemas en la red de Petri si hay transiciones en conflicto, dado que el test de habilitación es secuencial.

Una solución, mejor dicho, versión más correcta sería

```
if (M[P3]) {
        System.out.println (  "filósofo tres autorizado a pensar") ;

        M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
        }
```

Así se desmarcan las etapas de entrada antes de autorizar la ejecución de la transición.

La ejecución del código en este caso ofrece el siguiente resultado:

ramonpor:~/javahecho2/ java guardas5
transición 1 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 3 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)

transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
COORDINADOR FILÓSOFOS empieza ejecución
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo cinco autorizado a pensar
filósofo cinco pensando
final de 2 transiciones
false false true false false true false false true false false true false false true
filósofo uno termina de pensar
filósofo cinco termina de pensar
filósofo uno autorizado a comer
filósofo uno comiendo
final de 3 transiciones
false false false false false true false false true false false true false true false
filósofo tres termina de pensar
filósofo dos termina de pensar
filósofo cuatro termina de pensar
filósofo tres autorizado a comer
filósofo tres comiendo
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false true false false false false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo dos autorizado a comer
filósofo dos comiendo
final de 2 transiciones
false false false false false false false false true false true false false false false
filósofo uno termina de pensar
filósofo cinco termina de comer
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 1 transiciones
false true false false false false false false false false false false false false true

filósofo tres termina de pensar
final de 1 transiciones
false true false false false false false true false false false false false false true
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo uno autorizado a comer
filósofo uno comiendo
final de 1 transiciones
false false false false false true false true false false false false false false false
filósofo cinco termina de pensar
final de 1 transiciones
false false false false false true false true false false false false false true false
filósofo cuatro termina de comer
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false false false false false false false true false true false
filósofo dos termina de pensar
final de 1 transiciones
false false false false true false false false false false false true false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false false false false false false
filósofo cuatro termina de pensar
final de 1 transiciones
false false true false true false false false false false true false false false false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo dos autorizado a comer
filósofo dos comiendo
final de 2 transiciones
false false false false false false false false true false true false false false false
filósofo uno termina de pensar
filósofo cinco termina de comer
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 2 transiciones
false true false false false false false false false false false false false false true
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando

filósofo uno autorizado a comer
filósofo uno comiendo
final de 1 transiciones
false false false false false true false true false false false false false false false
filósofo cuatro termina de comer
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo tres autorizado a comer
filósofo tres comiendo

3.1.1 SELECCIÓN INDETERMINISTA Y RENDEZ-VOUS

El test de habilitación es secuencial, por lo tanto no es indeterminista. La primera condición del listado que se cumpla es la que se ejecuta. Esto puede dar lugar a que haya filósofos que coman poco.

```
if (M[T2] && M[T3] && M[C3]){
        System.out.println (  "filósofo tres autorizado a comer") ;

        M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
            }

            if   (M[T5] && M[T1] && M[C1]){
        System.out.println (  "filósofo uno autorizado a comer") ;

        M[T5]= false;M[T2]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
            }

            if   (M[T1] && M[T2] && M[C2]){
        System.out.println (  "filósofo dos autorizado a comer") ;

        M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
            }



            if   (M[T3] && M[T4] && M[C4]){
        System.out.println (  "filósofo cuatro autorizado a comer") ;

        M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
            }

            if   (M[T4] && M[T5] && M[C5]){
        System.out.println (  "filósofo cinco autorizado a comer") ;

        M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
            }
```

Haría falta que Java tuviera un select indeterminista como en ADA

Un ejemplo de lenguaje “JADA” sacado de Internet

```
class Coordinator extends ServerThread
{
public message int[] enterRead();
public message void  exitRead();
public message int[] enterWrite();
```

```
public message void  exitWrite();
}

class Buffer extends Coordinator
{
private int[] buff = new int[1000000];
Buffer()
 {
 super();
  for(int i=0; i < buff.length ; i++)
          buff[i] = 2*i;
 }

public void run()
 {
  int readers=0;
  for(;;)
  select
      {
          accept int[] enterRead()
          {
                  readers++;
                  reply(buff);
          }

          accept void exitRead()
          {
                  readers--;
                  reply();
          }

          accept int[] enterWrite()
          {
                  while(readers>0)
                  {
                          accept void exitRead()
                          {
                                  readers--;
                                  reply();
                          }
                  }
                  reply(buff);
                  accept void exitWrite()
                  {
                          reply();
                  }
          }
      }
  delay(15000)
      {
   System.out.println((new Date()).getTime()+" buffer : break");
          break;
      }
  System.out.println((new Date()).getTime()+" buffer : sali");
 }
}
```

Es una propuesta de incorporar al lenguaje JAVA:

Select indeterminista

Cita síncrona

3.1.2 MEJORA DEL COORDINADOR

Una posible mejora del coordinador es tratar primero la finalización de las transiciones y a continuación el test de habilitación

```
while(true){
//fin de transiciones
if (buffercoor.numberInBuffer>0) System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );

// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????

        while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
            switch(valor){
            case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
            case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
            case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
            case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
            case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
            case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
            case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
            case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
            case   TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println (      "filósofo cuatro termina de
comer");break;
            case   TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println (      "filósofo cinco termina de
comer");break;
            }// end swicth
            }//end while

// test de habilitación de transiciones
            if (M[P3]) {
            System.out.println (  "filósofo tres autorizado a pensar") ;

            M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
                }

            if (M[P1]) {
            System.out.println (  "filósofo uno autorizado a pensar") ;

            M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
                }

            if (M[P2]) {
            System.out.println (  "filósofo dos autorizado a pensar") ;

            M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
                }

            if (M[P4]) {
            System.out.println (  "filósofo cuatro autorizado a pensar") ;
```

```
M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
     }

if (M[P5]) {
System.out.println (  "filósofo cinco autorizado a pensar") ;

M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
     }
    if   (M[T2] && M[T3] && M[C3]){
System.out.println (  "filósofo tres autorizado a comer") ;

M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
     }

    if   (M[T5] && M[T1] && M[C1]){
System.out.println (  "filósofo uno autorizado a comer") ;

M[T5]= false;M[T2]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
     }

    if   (M[T1] && M[T2] && M[C2]){
System.out.println (  "filósofo dos autorizado a comer") ;

M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
     }



    if   (M[T3] && M[T4] && M[C4]){
System.out.println (  "filósofo cuatro autorizado a comer") ;

M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
     }

    if   (M[T4] && M[T5] && M[C5]){
System.out.println (  "filósofo cinco autorizado a comer") ;

M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
     }




   }//end while true
  } //en run

}// end coordinador
```

```
ramonpor:~/javahecho2/ java guardas7
transición 1 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 3 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
COORDINADOR FILÓSOFOS empieza ejecución
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo cinco autorizado a pensar
filósofo cinco pensando
final de 2 transiciones
false false true false false true false false true false false true false false true
filósofo uno termina de pensar
filósofo cinco termina de pensar
filósofo uno autorizado a comer
filósofo uno comiendo
final de 3 transiciones
false false false false false true false false true false false true false true false
filósofo tres termina de pensar
filósofo dos termina de pensar
filósofo cuatro termina de pensar
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false true false false false false false true true false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false true false false false false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo dos autorizado a comer
filósofo dos comiendo
final de 1 transiciones
false false false false false false false false true false true false false false false
filósofo uno termina de pensar
```

final de 2 transiciones
false true false false false false false true false true false false false false
filósofo tres termina de pensar
filósofo cinco termina de comer
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 1 transiciones
false true false false false false false true false false false false false false true
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo uno autorizado a comer
filósofo uno comiendo
final de 1 transiciones
false false false false false true false true false false false false false false false
filósofo cinco termina de pensar
final de 2 transiciones
false false false false false true false true false false false false false true false
filósofo dos termina de pensar
filósofo cuatro termina de comer
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo tres autorizado a comer
filósofo tres comiendo
final de 1 transiciones
false false false false true false false false false false false true false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo cinco autorizado a comer
filósofo cinco comiendo
final de 1 transiciones
false false true false true false false false false false false false false false false
filósofo cuatro termina de pensar
final de 2 transiciones
false false true false true false false false false false true false false false false
filósofo cinco termina de comer
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo dos autorizado a comer
filósofo dos comiendo
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
final de 1 transiciones
false false false false false false false false false false false false false false true
filósofo uno termina de pensar
final de 3 transiciones

false true false false false false false false false false false false false true
filósofo cinco termina de pensar
filósofo dos termina de comer
filósofo cuatro termina de comer
filósofo dos autorizado a pensar
filósofo dos pensando
filósofo cuatro autorizado a pensar
filósofo cuatro pensando
filósofo uno autorizado a comer
filósofo uno comiendo
final de 1 transiciones
false false false false false true false false true false false true false true false
filósofo tres termina de pensar
filósofo tres autorizado a comer
filósofo tres comiendo

3.1.3 PLANIFICACIÓN DE HILOS EN JAVA CLÁSICO

Se observa un problema: la "inversión de prioridad" que se manifiesta. El coordinador debería ejecutarse sin interrupción por parte de las tareas de prioridad inferior. Cuando el coordinador desbloquea un monitor el procesador salta a la tarea desbloqueada. Realmente no es una "inversión de prioridad" sino que el planificador de java reparte tiempos de ejecución del procesador entre los diferentes hilos del programa. El java en linux utiliza un sistema de planificación no preemptivo

Ejemplo:

COORDINADOR FILÓSOFOS empieza ejecución	
filósofo tres autorizado a pensar	(coordinador prioridad 10)
filósofo tres pensando	(transición prioridad 5)
filósofo uno autorizado a pensar	(coordinador prioridad 10)
filósofo uno pensando	(transición prioridad 5)
filósofo dos autorizado a pensar	(coordinador prioridad 10)
filósofo dos pensando	(transición prioridad 5)
filósofo cuatro autorizado a pensar	(coordinador prioridad 10)
filósofo cuatro pensando	(transición prioridad 5)
filósofo cinco autorizado a pensar	(coordinador prioridad 10)
filósofo cinco pensando	(transición prioridad 5)

Cuando el thread transicion recibe la autorización del coordinador el sistema de ejecución de tareas de java hace que el thread transición ocupe el procesador aunque tenga menos prioridad (5) que el coordinador (10).

"A pesar de ello" la evolución de la red de Petri es correcta

Para que las transiciones cedan el procesador al coordinador, de forma que este pueda acabar el test de habilitación sin interrupción, se ha añadido en la clase transicion la siguiente instrucción (en negrita):

```
class transicion extends Thread{
private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
```

```
transicion(int aidentidad,comunica ac,BoundedBuffer abuffer){
   identidad=aidentidad;
   bufferproceso = abuffer;
   c =ac;
  }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){          //System.out.println (identidad + " esperando autorización") ;
   c.preguntautorizacion();
   this.yield();
switch(identidad){
          case TF1_P:System.out.println ("filósofo uno pensando") ;break;
          case TF2_P:System.out.println ("filósofo dos pensando") ;break;
          case TF3_P:System.out.println ("filósofo tres pensando") ;break;
          case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
          case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
          case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
          case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
          case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
          case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
          case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
          }

          for (i = 0; i < 100000000; i++) {
          // comiendo o pensando
          }


   bufferproceso.put(identidad);

  }
  }
}
```

Mediante la instrucción **this.yield()** se obliga al thread a pausar su ejecución y ceder el procesador a las tareas de mayor prioridad que sean ejecutables en ese momento.

La ejecución del programa produce la siguiente traza:

```
ramonpor:~/javahecho2/ java guardas5
transición 1 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 3 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
COORDINADOR FILÓSOFOS empieza ejecución
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a pensar
```

filósofo cuatro autorizado a pensar
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro pensando
filósofo dos pensando
filósofo uno pensando
filósofo tres pensando
final de 5 transiciones
false false true false false true false false true false false true false false true
filósofo cuatro termina de pensar
filósofo dos termina de pensar
filósofo uno termina de pensar
filósofo cinco termina de pensar
filósofo tres termina de pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo uno comiendo
filósofo tres comiendo
final de 1 transic iones
false false false false true false false false false false true true false true false
filósofo tres termina de comer
filósofo tres autorizado a pensar
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
final de 1 transiciones
false false false false true true false false false false false false false true false
filósofo uno termina de comer
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo dos comiendo
filósofo uno pensando
final de 2 transiciones
false false false false false false false false false false false false false true true
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo cuatro autorizado a pensar
filósofo cinco autorizado a comer
filósofo cinco comiendo
filósofo cuatro pensando
final de 2 transiciones
false false false false false false false true true false false false false false false
filósofo dos termina de comer
filósofo uno termina de pensar
filósofo dos autorizado a pensar
filósofo tres autorizado a comer
filósofo tres comiendo
filósofo dos pensando
final de 1 transiciones
false true true false false false false false false false false false false false false
filósofo cuatro termina de pensar
final de 3 transiciones

false true true false false false false false false false true false false false false
filósofo dos termina de pensar
filósofo tres termina de comer
filósofo cinco termina de comer
filósofo tres autorizado a pensar
filósofo cinco autorizado a pensar
filósofo uno autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo uno comiendo
filósofo cinco pensando
filósofo tres pensando
final de 3 transiciones
false false false false true true false false false false false false false false false
filósofo tres termina de pensar
filósofo uno termina de comer
filósofo cinco termina de pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo dos comiendo
filósofo uno pensando
filósofo cuatro termina de comer
filósofo cuatro autorizado a pensar
filósofo cinco autorizado a comer
filósofo cinco comiendo
filósofo cuatro pensando
final de 1 transiciones
false false false false false false false true true false false false false false false
filósofo uno termina de pensar
final de 3 transiciones
false true false false false false false true true false false false false false false
filósofo cuatro termina de pensar
filósofo cinco termina de comer
filósofo dos termina de comer
filósofo dos autorizado a pensar
filósofo cinco autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo uno comiendo
filósofo tres comiendo
filósofo cinco pensando
filósofo dos pensando

Evolución de la red de Petri

P1	C1	T1	P2	C2	T2	P3	C3	T3	P4	C4	T4	P5	C5	T5
true	false	true	true	false	true	true	false	true	true	false	true	true	false	true
false	false	true	false	false	true	false	false	true	false	false	true	false	false	true
false	false	false	false	true	false	false	false	false	false	true	true	false	true	false
false	false	false	false	true	true	false	false	false	false	false	false	false	true	false
false	false	false	false	false	false	false	false	false	false	false	false	false	true	true
false	false	false	false	false	false	false	true	true	false	false	false	false	false	false
false	true	true	false	false	false	false	false	false	false	false	false	false	false	false
false	true	true	false	false	false	false	false	false	false	true	false	false	false	false
false	false	false	false	true	true	false	false	false	false	false	false	false	false	false
false	false	false	false	false	false	false	true	true	false	false	false	false	false	false
false	true	false	false	false	false	false	true	true	false	false	false	false	false	false

El estado de la red de Petri se imprime después del test de habilitación y previamente a la finalización de las transiciones pero solo si se produce algún fin de transición.

3.2 RED COMÚN RESUELTA CON EL PROBLEMA DE LAS GUARDAS

```
class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }

     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }

     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}

class comunica {
// monitores utilizados por el coordinador para autorizar la ejecución
// de transciones
```

```
private boolean disponible = false;

public synchronized void preguntautorizacion()
        {
         while(!disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
                disponible=false;
                notify();

        }
public synchronized void autorizoejecucion()
        {
                while(disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                    }

                        disponible=true;
                        notify();
        }
}

class transicion extends Thread{
private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int T1=0,T2=1,T3=2,T4=3,T5=4,T6=5,T7=6,T8=7,T9=8,T10=9,T11=10;

transicion(int aidentidad,comunica ac,BoundedBuffer abuffer){
         identidad=aidentidad;
         bufferproceso = abuffer;
         c =ac;
        }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){//System.out.println (identidad + " esperando autorización") ;
        c.preguntautorizacion();
switch(identidad){
                case T1:System.out.println ("transición uno dispara") ;break;
                case T2:System.out.println ("transición dos dispara") ;break;
                case T3:System.out.println ("transición tres dispara") ;break;
                case T4:System.out.println ("transición cuatro dispara") ;break;
                case T5:System.out.println ("transición cinco dispara") ;break;
                case T6:System.out.println ("transición seis dispara") ;break;
                case T7:System.out.println ("transición siete dispara") ;break;
                case T8:System.out.println ("transición ocho dispara") ;break;
                case T9:System.out.println ("transición nueve dispara") ;break;
```

```
            case T10:System.out.println ("transición diez dispara") ;break;
            case T11:System.out.println ("transición once dispara") ;break;
                }

            for (i = 0; i < 100000000; i++) {
            // transciones ejecutando código
                }


        bufferproceso.put(identidad);

       }
       }
}

class coordinador extends Thread{
private String nombre;
final int P1 =0,P2=1,P3=2,P4=3,P5=4,P6=5,P7=6,P8=7,P9=8,P10=9,P11=10,P12=11,R11=12,R12=13,R21=14,R22=15;
final int T1=0,T2=1,T3=2,T4=3,T5=4,T6=5,T7=6,T8=7,T9=8,T10=9,T11=10;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];

private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
        nombre=anombre;
        vectorcomunica=avectorcomunica;
        M=aestado;
        buffercoor=abuffercoor;
       }

public void run(){
System.out.println (nombre + " empieza ejecución");

while(true){
            if (M[P1]) {
            System.out.println (  "AUTORIZO EJECUCIÓN T1") ;

            vectorcomunica[T1].autorizoejecucion();M[P1]= false;
                }

            if (M[P2]) {
            System.out.println (  "AUTORIZO EJECUCIÓN T2") ;

            vectorcomunica[T2].autorizoejecucion();M[P2]= false;
                }

            if (M[P4] && M[R11]) {
            System.out.println (  "AUTORIZO EJECUCIÓN T4") ;

            vectorcomunica[T4].autorizoejecucion();M[P4]= false;M[R11]=false;
                }

            if (M[P4] && M[R12]) {
```

```
        System.out.println (  "AUTORIZO EJECUCIÓN T3") ;

        vectorcomunica[T3].autorizoejecucion();M[P4]= false;M[R12]=false;
            }

        if (M[P5] && M[P7]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T5") ;

        vectorcomunica[T5].autorizoejecucion();M[P5]= false;M[P7]=false;
            }

        if (M[P8] && M[R22]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T6") ;

        vectorcomunica[T6].autorizoejecucion();M[P8]= false;M[R22]=false;
            }

        if (M[P3] && M[P9]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T7") ;

        vectorcomunica[T7].autorizoejecucion();M[P3]= false;M[P9]=false;
            }

        if (M[P8] && M[R21]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T8") ;

        vectorcomunica[T8].autorizoejecucion();M[P8]= false;M[R21]=false;
            }

        if (M[P6]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T9") ;

        vectorcomunica[T9].autorizoejecucion();M[P6]= false;
            }

        if (M[P10]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T10") ;

        vectorcomunica[T10].autorizoejecucion();M[P10]= false;
            }
        if (M[P11] && M[P12]) {
        System.out.println (  "AUTORIZO EJECUCIÓN T11") ;

        vectorcomunica[T11].autorizoejecucion();M[P11]= false;M[P12]=false;
            }




    while(buffercoor.numberInBuffer>0) {
        valor= buffercoor.get();
    switch(valor){
        case T1:M[P2]=true;M[P3]=true;System.out.println (  "transición uno finalizada") ;break;
        case T2:M[P4]=true;System.out.println (  "transición dos finalizada") ;break;
        case T3:M[P5]=true;M[R11]=true;System.out.println (  "transición tres finalizada") ;break;
```

```
            case T4:M[P2]=true;M[R12]=true;System.out.println (  "transición cuatro finalizada") ;break;
            case T5:M[P1]=true;System.out.println (  "transición cinco finalizada") ;break;
            case T6:M[P9]=true;M[R21]=true;System.out.println (  "transición seis finalizada") ;break;
            case T7:M[P6]=true;M[P10]=true;System.out.println (  "transición siete finalizada") ;break;
            case T8:M[R22]=true;M[P11]=true;M[P12]=true;System.out.println (  "transición ocho finalizada") ;break;
            case T9:M[P7]=true;M[P11]=true;System.out.println (  "transición nueve finalizada") ;break;
            case T10:M[P12]=true;System.out.println (  "transición diez finalizada") ;break;
            case T11:M[P8]=true;System.out.println (  "transición once finalizada") ;break;
            }// end swicth


        }//end while
        }//end while true
       } //en run

}// end coordinador


class guardas6{


public static void main (String args[]){
   int i;

// definición de contantes para las transiciones
final int T1=0,T2=1,T3=2,T4=3,T5=4,T6=5,T7=6,T8=7,T9=8,T10=9,T11=10;

// definición de contantes para los lugares

final int P1 =0,P2=1,P3=2,P4=3,P5=4,P6=5,P7=6,P8=7,P9=8,P10=9,P11=10,P12=11,R11=12,R12=13,R21=14,R22=15;

// estado incial de la red de petri
boolean[] estado={true,false,false,false,false,false,false,true,false,false,false,false,true,false,true,false};

// vector de monitores para comunicar la acepatación del comienzo del disparo de las transiciones


comunica com1=new comunica();
comunica com2=new comunica();
comunica com3=new comunica();
comunica com4=new comunica();
comunica com5=new comunica();
comunica com6=new comunica();
comunica com7=new comunica();
comunica com8=new comunica();
comunica com9=new comunica();
comunica com10=new comunica();
comunica com11=new comunica();


comunica[] comunicaarray = {com1,com2,com3,com4,com5,com6,com7,com8,com9,com10,com11};



// buffer para comunicar los fin de code
// hay once transiciones aunque no puedan acabar todas de golpe, le damos tamaño once
```

```
BoundedBuffer buffer= new BoundedBuffer(11);



transicion T1task= new transicion(T1,comunicaarray[0], buffer);
transicion T2task= new transicion(T2,comunicaarray[1], buffer);
transicion T3task= new transicion(T3,comunicaarray[2], buffer);
transicion T4task= new transicion(T4,comunicaarray[3], buffer);
transicion T5task= new transicion(T5,comunicaarray[4], buffer);
transicion T6task= new transicion(T6,comunicaarray[5], buffer);
transicion T7task= new transicion(T7,comunicaarray[6], buffer);
transicion T8task= new transicion(T8,comunicaarray[7], buffer);
transicion T9task= new transicion(T9,comunicaarray[8], buffer);
transicion T10task= new transicion(T10,comunicaarray[9], buffer);
transicion T11task= new transicion(T11,comunicaarray[10], buffer);



transicion[] Transicionarray= {T1task,T2task,T3task,T4task,T5task,T6task,T7task,T8task,T9task,T10task,T11task};

for (i = 0; i < 11; i++) {
            Transicionarray[i].start();
           }

coordinador coor=new coordinador("COORDINADOR COMÚN ",comunicaarray,estado,buffer);
coor.setPriority(10);

coor.start();

}
}
```

```
ramonpor:~/javahecho2/ java guardas6
transición 1 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 3 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
transición 11 empieza a ejecutarse (start)
COORDINADOR COMÚN  empieza ejecución
AUTORIZO EJECUCIÓN T1
AUTORIZO EJECUCIÓN T8
transición uno dispara
transición ocho dispara
transición uno finalizada
AUTORIZO EJECUCIÓN T2
transición dos dispara
transición ocho finalizada
AUTORIZO EJECUCIÓN T11
transición once dispara
transición dos finalizada
AUTORIZO EJECUCIÓN T4
transición cuatro dispara
transición once finalizada
AUTORIZO EJECUCIÓN T6
transición seis dispara
transición cuatro finalizada
AUTORIZO EJECUCIÓN T2
transición dos dispara
transición seis finalizada
AUTORIZO EJECUCIÓN T7
transición siete dispara
transición dos finalizada
AUTORIZO EJECUCIÓN T3
transición tres dispara
transición siete finalizada
AUTORIZO EJECUCIÓN T9
transición nueve dispara
AUTORIZO EJECUCIÓN T10
transición diez dispara
transición tres finalizada
transición nueve finalizada
AUTORIZO EJECUCIÓN T5
transición cinco dispara
transición diez finalizada
AUTORIZO EJECUCIÓN T11
transición once dispara
transición cinco finalizada
AUTORIZO EJECUCIÓN T1
```

transición uno dispara
transición once finalizada
AUTORIZO EJECUCIÓN T8
transición ocho dispara

JAVA ESPECIFICACION PARA TIEMPO REAL

4.1 INTRODUCCIÓN

La característica principal de todo sistema en tiempo real es que sus acciones deben ejecutarse en intervalos de tiempo determinados por la dinámica de los sistemas físicos que controlan. Por ello existe una necesidad de fiabilidad y seguridad en la ejecución. El problema de los sistemas empotrados (sistema electrónico que realiza un tarea específica bajo la supervisión de un microprocesador y un programa computacional) y que comparten recursos de sistema con otros sistemas que puedan estar conectados al sistema supervisor es que están limitados. La mayor parte de estos procesos del sistema supervisor se ejecutan de forma concurrente. Ello quiere decir que intentan aportar una sensación de operación de forma paralela (llevar a cabo varias tareas a la vez) sobre el microprocesador (monoprocesador), aunque ello no sea realmente así ya que el micro solo puede llevar a cabo un único proceso a la vez. Con todo se observa la necesidad de una ejecución predecible en estos sistemas además de una reducción de tiempos de validación de datos.

A la hora de usar java para programar tiempo real surgen problemas tales como una dificultad de obtener predecibilidad en los resultados debido al recolector de basura, el cual es un proceso, ejecutado cuando el sistema "determina" necesario liberar memoria. También reduce eficiencia ya que es un lenguaje (compilado-**interpretado).** Con el compilado se convierte el código fuente que reside en archivos cuya extensión es **.java,** a un conjunto de instrucciones que recibe el nombre de *bytecodes* que se guardan en un archivo cuya extensión es **.class**. Estas instrucciones son independientes del tipo de ordenador. La interpretación ocupa tiempo de procesamiento.

Cada intérprete Java es una implementación de la Máquina Virtual Java (JVM). Los *bytecodes* posibilitan el objetivo de WORA "write once, run anywhere", de escribir el programa una vez y que se pueda correr en cualquier plataforma que disponga de una implementación de la JVM. Por ejemplo, el mismo programa Java puede correr en Windows 98, Solaris, Macintosh, etc.

La Máquina Virtual Java (JVM) es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un ordenador

abstracto y especifica las instrucciones (*bytecodes*) que este ordenador puede ejecutar. El intérprete Java específico ejecuta las instrucciones que se guardan en los archivos cuya extensión es **.class**. Las tareas principales de la JVM son las siguientes:

Reservar espacio en memoria para los objetos creados

Liberar la memoria no usada (garbage collection).

Asignar variables a registros y pilas

Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos

Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java

Teniendo todo lo anterior y la necesidad de crear un programa que trabaje en tiempo real se crea la especificación para tiempo real de java la cual es compatible a entornos java además de serlo con las versiones anteriores. Soporta una ejecución predecible. Permite variaciones en las decisiones de implementación.

En la especificación de java para tiempo real no hay extensiones sintácticas (ejemplo la sintaxis de la cláusula synchronized permanecen invariable aunque su interpretación difiera). Con cada una de las versiones que Sun lanza del JDK (Kit Development Java), se acompaña de una serie de bibliotecas con clases estándar que valen como referencia para todos los programadores en Java. Estas clases se pueden incluir en los programas Java, sin temor a fallos de portabilidad. Además, están bien documentadas (mediante páginas Web), y organizadas en paquetes y en un gran árbol de herencia. A este conjunto de paquetes (o bibliotecas) se le conoce como la API de Java (Application Programming Interface).

4.2 JRATE

Para subsanar el problema de la no predecibilidad en la ejecución de las tareas en los programas java, dado que depende de la política aplicada por el scheduler de java, se decide implementar los programas en java para tiempo real.

Esto conlleva alguna limitación dado que librerías de Java para tiempo real sólo existen, por ahora, para el sistema operativo Linux.

Se utiliza la plataforma de programación jRate cuyo autor es Angelo Corsaro.

jRate es una implementación de código abierto de la extensión de Java para tiempo real que ha sido desarrollada en la Universidad de California Irvine. jRate extiende las funcionalidades del compilador GNU de código abierto para Java. Proporciona un sistema de compilación previo para las aplicaciones que cumplan con la extensión de Java tiempo real.

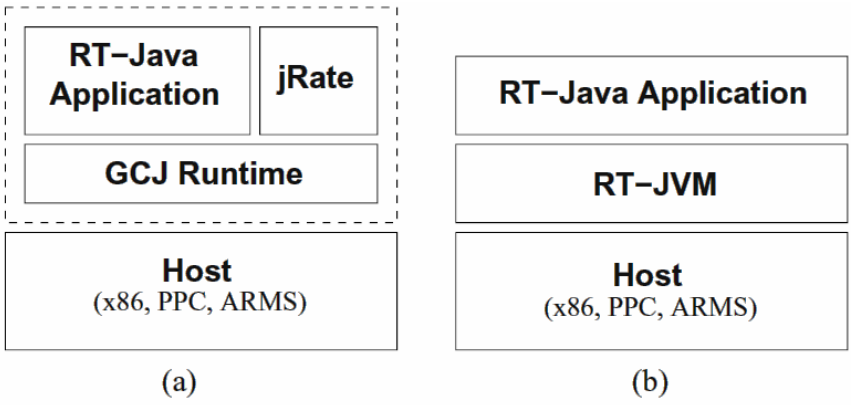

Figura 6 Estructura de jRate

jRate no produce bytecodes para ejecutar en una máquina virtual Java .jRate produce código nativo para la plataforma de ejecución. El acceso a los servicios de Java y Java para tiempo real, como puede ser el recolector de basura, hilos de tiempo real, y el scheduler, se produce mediante las librerías de sistema (Runtime systems) de GCJ y jRate respectivamente.

Un inconveniente de este sistema de trabajo es que puede dificultar la transportabilidad de las aplicaciones que deberán ser recompiladas en nuevas arquitecturas.

En la práctica esto se ve compensado debido a que las aplicaciones se compilan a código nativo y esto supone una gran ventaja en comparación con plataformas de Java basadas en interpretes (clásicas máquinas virtuales java) o en compiladores just in time.

4.3 THREADS EN TIEMPO REAL

La clase Realtimethread deriva de la clase java.lang.Thread, por lo tanto incluye sus métodos y su forma de utilizarla es similar; se deberá crear una clase que derive de Realtimethread e incluir en ella un método run() con las instrucciones que se deben ejecutar.

El constructor de la clase es:

RealtimeThread ([SchedulingParameters scheduling,] [ReleaseParameters release,][MemoryParameters memory, MemoryArea area,] [ProcessingGroupParameters]);

A continuación se detallan los diferentes tipos de parámetros

4.3.1 SCHEDULINGPARAMETERS

Son los parámetros utilizados por el Scheduler, su subclase más importante es PriorityParameters que sirve para definir la prioridad que se le asigna al thread.

Constructor
`SchedulingParameters()`

Clase derivada: PriorityParameters

public class PriorityParameters extends SchedulingParameters();
Constructor: PriorityParameters(int prioridad): define el nivel de prioridad;

Constructor
`PriorityParameters(int priority)` Crea una instancia de SchedulingParameters con la prioridad dada.

Métodos	
`int`	`getPriority()` devuelve el valor de la prioridad.
`void`	`setPriority(int priority)` establece el valor de la prioridad.
`java.lang.String`	`toString()`

4.3.2 RELEASEPARAMETERS

Son los parámetros relativos al comportamiento del thread, dependiendo del tipo de thread que se use (periódico, esporádico, etc.) se utiliza una u otra de sus subclases.

Constructor

ReleaseParameters (RelativeTime coste, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);

Coste: tiempo de cpu utilizado por el thread.

Deadline: plazo de respuesta.

OverrunHandler: Thread gestor del desbordamiento en el coste indicado.

MissHandler: Thread gestor del desbordamiento en el deadline indicado.

Métodos	
RelativeTime	**getCost**() Devuelve el valor del tiempo de computo.
AsyncEventHandler	**getCostOverrunHandler**() devuelve un puntero al thread overrun handler.
RelativeTime	**getDeadline**() Devuelve el valor del deadline.
AsyncEventHandler	**getDeadlineMissHandler**() devuelve un puntero al thread miss handler.
void	**setCost**(RelativeTime cost) Establece el valor del costo.
void	**setCostOverrunHandler**(AsyncEventHandler handler) establece un puntero al thread overrun handler.
void	**setDeadline**(RelativeTime deadline) Establece el valor del deadline.
void	**setDeadlineMissHandler**(AsyncEventHandler handler) establece un puntero al thread deadlinemisshandler.
boolean	**setIfFeasible**(RelativeTime cost, RelativeTime deadline) Después de considerar el valor de los parámetros, devuelve verdad si el thread es planificable

Las clases derivadas de ReleaseParameters son:

PeriodicParameters:

Para especificar que un thread se ejecute de forma periódica.

PeriodicParameters (HighResolutionTime start, RelativeTime periodo, RelativeTime coste, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);

Constructor
PeriodicParameters(HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler) Crea una instancia de PeriodicParameters.

Method Summary	
RelativeTime	**getPeriod**() Devuelve el valor del periodo.
HighResolutionTime	**getStart**() Devuelve el valor del tiempo de inicioet the start time.
boolean	**setIfFeasible**(RelativeTime period, RelativeTime cost, RelativeTime deadline) Después de considerar el valor de los parámetros, devuelve verdad si el thread es planificable
void	**setPeriod**(RelativeTime period) Set the period value.
void	**setStart**(HighResolutionTime start) Set the start time.

Un ejemplo de uso de PeriodicParameters es el siguiente:

```
public Señaldigital extends RealTimeThread{
public Señaldigital(AsyncEventHandler overrun, AsyncEventHandler miss) {
  super();
  setReleaseParameters(
     new PeriodicParameters(
new RelativeTime(0,0),          /* start */
new RelativeTime(1000,0),       /* period */
new RelativeTime(5,0),          /* cost */
new RelativeTime(500,0),        /* deadline o plazo */
overrun, miss                   /* handlers */
)
);
```

```
}
}
```

AperiodicParameters:

Caracteriza un objeto planificable que puede hacerse activo en cualquier momento..

Constructor
`AperiodicParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)`

SporadicParameters

Para tareas esporádicas (existe un tiempo mínimo entre 2 activaciones consecutivas). La clase SporadicParameters añade como primer parámetro (a los de ReleaseParameters) el Relativetime intervalo mínimo que refleja el tiempo mínimo entre 2 activaciones consecutivas.

ProcessingGroupParameters

Definen un grupo de tareas aperiódicas que no podrán consumir más de un determinado tiempo en cada periodo de ejecución indicado.

Parámetros de Memoria

MemoryParameters y MemoryArea definen la utilización de la memoria que realizará el thread.

4.4 PLANIFICACIÓN

La planificación consiste en la asignación de recursos y tiempo a actividades de forma que se cumplan determinados requisitos impuestos de eficiencia. Estos requisitos dependen del sistema computacional, dando prioridad en los sistemas que no son de tiempo real, a los tiempos de ejecución y tiempos de respuesta principalmente. Sin embargo, los sistemas de tiempo real se centran en el cumplimiento de los tiempos que se hayan especificado para la actividad determinada.

A la hora de realizar el método de planificación, hay que tener en cuenta dos aspectos que han de conformarlo. En un principio hay que determinar el orden de acceso de las tareas a los recursos dependiendo de las prioridades de cada una mediante un algoritmo planificador. Por otra parte también hay que determinar el comportamiento temporal del sistema mediante un método de análisis, esto se realiza mediante la comprobación de que los requisitos temporales están garantizados en todos los casos posibles, además del estudio del peor caso posible. Estos algoritmos pueden ser estáticos o dinámicos. Para una planificación estática los requisitos son analizados antes de la ejecución y se determina el orden en que estas se ejecutarán. Ello provoca una menor flexibilidad, aunque se reducen los tiempos de ejecución. Para una planificación dinámica el orden de las tareas es decidido durante la ejecución, además de realizarse también su análisis.

En un STR (sistema de tiempo real) la planificación de las tareas concurrentes (al contrario que en un STC(Sistema de tiempo compartido) es algo importante que se debe cumplir, y debe asegurar:

1. **Garantía de plazos**: Un STR funciona correctamente cuando se garantizan los plazos de todas las tareas. En un STC lo importante es asegurar un flujo lo más elevado posible.

2. **Estabilidad**: Caso de sobrecarga, un STR debe garantizar que al menos un subconjunto de tareas cumplen sus plazos (tareas *críticas*). En un STC, hay que repartir equitativamente el tiempo de ejecución.

3. **Tiempo de respuesta máximo**: En un STR se trata de acotar el tiempo de respuesta máximo de las tareas. En un STC se trata de minimizar el tiempo de respuesta medio.

4.4.1 PLANIFICACIÓN EN JAVA PARA TIEMPO REAL

Existe una gran variedad de algoritmos de planificación adecuados para distintos tipos de sistemas: tiempo real crítico, multimedia, etc. El objetivo es escribir una especificación que se adapte a los algoritmos de planificación proporcionados por los SO existentes, por ejemplo, algoritmo de planificación estático RMS (Rate Monotonic Scheduling). La planificación monótona de frecuencia es uno de los métodos con mayor perspectiva para la resolución de conflictos de la planificación multitarea con tareas periódicas. Lo que hace el RMS es asignar prioridades a las tareas dependiendo de sus periodos, a menor periodo mayor prioridad. Se deberá estructurar las tareas fundamentales para que tengan periodos cortos o modificando las prioridades del RMS para que tengan en cuenta dichas tareas. Otro objetivo es realizar una especificación capaz de adaptarse a futuros algoritmos por ejemplo, algoritmo de planificación dinámico EDF (Earliest Deadline First). EDF es un algoritmo dinámico expulsivo basado en prioridades.

RTJ (Real Time Java), tiene al menos 28 niveles de prioridad, permitiendo definir algoritmos de planificación. RTJ mantiene compatibilidad hacia atrás con la planificación existente en Java.

En la especificación de tiempo real de Java el concepto de ser planificable es un atributo del objeto, con interfaz *Schedulable* o planificable. Cualquier objeto que implemente esta interfaz es planificado por el planificador (*threads* de tiempo real y manejadores de eventos asíncronos). Se pueden definir varios tipos de *scheduler* que tendrán como nombre el tipo de política aplicable seguida de la palabra *Scheduler* (ejemplo EDFScheduler). La clase genérica de planificadores es class Scheduler. La cual contiene los métodos para admisión de *threads*, mecanismos de manejo de eventos asíncronos, etc. Las subclases derivadas de *Scheduler* implementan algoritmos alterativos de planificación.

Los *Scheduler* se encargan de verificar si las tareas que deben planificar son factibles de acuerdo con la política utilizada. Gestionar la ejecución de objetos planificables (interface Schedulable).

El *Scheduler* básico es el PriorityScheduler, el cual realiza la planificación por prioridades fijas, esto quiere decir que la prioridad de un objeto planificable no cambia, excepto por el protocolo de control de inversión de prioridades. Utiliza planificación expulsiva: si en algún momento se activa un objeto planificable de prioridad superior al que está actualmente en ejecución, este último es expulsado del procesador. Existen al menos 28 niveles de prioridad para *threads* de tiempo real.

Existen 3 tipos de objetos o tareas planificables:

- RealtimeThread ***(threads* de tiempo real)**: Menor prioridad que el garbage colector (GC).

- NoHeapRealtimeThread ***(threads* de tiempo real)**: Similar al anterior pero no coloca objetos en el heap y por tanto puede ejecutar siempre antes que el recolector de basura.
- AsyncEventHandler **(manejadores de eventos asíncronos)**: Gestor de eventos asíncronos.

En RT-Java el algoritmo de planificación, por defecto, será preemptivo (con desalojo) y con prioridades fijas. La política que se utiliza para comprobar si es factible una determinada planificación de tareas será, por defecto, RMA (Rate Monotonic Analisys). Esta política nos permite analizar el comportamiento temporal de las aplicaciones de tiempo real basadas en prioridades fijas.

4.4.2 SCHEDULER

Scheduler es una clase abstracta que gestiona la ejecución de objetos planificables e implementa algoritmos de factibilidad. Se encarga en general de manipular el objeto planificable a la hora de trabajar con el planificador y de determinar todo con lo que respecta a la planificación utilizada.

Una instancia de scheduler maneja la ejecución de objetos planificables y pone en práctica un algoritmo de factibilidad. El algoritmo de factibilidad determina si, conocido el conjunto de objetos planificables y sus prioridades, se puede ejecutar su planificación.

El creador de la plataforma de Java tiempo real puede definir diferentes subclases de Scheduler que serán usadas en políticas alternativas de planificación. jRate sólo define la subclase PriorityScheduler

Métodos básicos	
`void`	addToFeasibility (Schedulable schedulable) : añade el objeto schedulable al planificador.
`void`	removeToFeasibility (Schedulable schedulable): elimina el objeto schedulable del planificador.
`abstract boolean`	**`isFeasible`**`()` true si los objetos planificables con los que cuenta (añadidos con addToFeasibilitiy()) permiten una planificación factible.
java.lang.String	getPolicyName(): Devuelve el nombre de la política de planificación utilizada.
void	setDefaultScheduler (Scheduler scheduler): define el scheduler indicado como el utilizado por defecto.
Scheduler	getDefaultScheduler (Scheduler scheduler): devuelve una referencia al scheduler por defecto, que si no se ha definido lo contrario será un PriorityScheduler.

Un ejemplo de su uso sería el siguiente en el cual se suponen 3 señales como la expuesta en el ejemplo de PeriodicParameters, con valores diferentes de parámetros:

```
Señaldigital1 S1 = new Señaldigital1();
Señaldigital2 S2 = new Señaldigital2 ();
Señaldigital3 S3 = new Señaldigital3 ();
Scheduler plani = Scheduler.getDefaultScheduler();
plani.addToFeasibility(S1);
plani.addToFeasibility(S2);
plani.addToFeasibility(S3);
if (!plani.isFeasibile())
throw new Error("Cannot accommodate threads");
```

PRIORITYSCHEDULER

La clase Priorityscheduler es un Scheduler preemptivo con prioridades fijas.

Sus métodos básicos:

Métodos básicos	
`void`	fireSchedulable (Schedulable schedulable): Dispara la ejecución del objeto schedulable.
`int`	getxxxPriority() Dependiendo de si xxx es Max, Min o Norm devolverá la máxima, mínima o normal prioridad disponible para un thread gestionado por este scheduler.
`int`	getxxxPriority(java.lang.Thread thread) Si el thread indicado es planificable por el scheduler y dependiendo de si xxx es Max, Min o Norm devolverá la máxima, mínima o normal prioridad que puede tener un thread gestionado por este scheduler
`PriorityScheduler`	instance(): devuelve un puntero a una instancia de un PriorityScheduler

4.5 SINCRONIZACIÓN

Java para tiempo real mantiene su compatibilidad hacia atrás. Para la sincronización se utiliza la palabra reservada *synchronized* de Java (monitores). Como diferencias con respecto al java clásico a la hora de acceder a un bloque sincronizado existe un determinismo haciendo que los *threads* estén ordenados por prioridad. Dado el caso de que un *thread* bloqueado pase a estar preparado, este se pone como último de la cola de su prioridad. Los threads de igual prioridad se ordenan por FIFO.

Se deben utilizar mecanismos adicionales para evitar el bloqueo con threads de menor prioridad que el GC.

La cláusula synchronized, que protege el acceso a recursos compartidos puede provocar el típico problema de los STR de inversión de prioridad. Para subsanarlo, RTSJ modifica la interpretación de dicha cláusula definiendo MonitorControl como clase base para la política de elección threads cuando

puede darse ese problema de forma que se evite la inversión de prioridad. Cuando varios threads acceden a una zona sincronizada se aplica, por defecto, una política de herencia de prioridad *(PriorityInheritance)*, pero es posible utilizar también un protocolo de emulación de techo de prioridad (*PriorityCeilingEmulation).*

donde el protocolo por defecto es el de herencia de prioridad. Se pueden establecer otros algoritmos para todos o para un monitor particular, como.

4.5.1 MONITORCONTROL

Es la clase base para definir de la política de control de monitores. Parece ser que no esta implementada en jRate

Constructor	
`protected`	`MonitorControl()`.

Métodos	
`void`	*setMonitorControl(MonitorControl politica);* define la política aplicable cuando se accede a recursos comunes (synchronize)
`void`	*setMonitorControl(java.lang.Object monitor, MonitorControl politica):* define la política aplicable cuando se accede a recursos comunes, sólo para el objeto indicado.

Sus subclases son:

• PriorityCeilingEmulation: protocolo de techo de prioridad

public PriorityCeilingEmulation(int ceiling): crea un objeto de esta clase con el techo indicado.

• PriorityInheritance: protocolo de herencia de prioridad

- *public PriorityInheritance()*

4.6 EVENTOS ASÍNCRONOS

La mayoría de eventos del entorno son asíncronos a la ejecución del sistema. Estos eventos asíncronos también pueden aparecer internamente tal como podría ser de la Java Virtual Machine o programables por la aplicación

La forma de tratarlos es mediante un mecanismo dado por esta especificación de java para unir la ocurrencia de un evento a un manejador (objeto planificable). La ocurrencia del evento hace que el objeto asociado a ese evento sea planificado para ejecución. Una instancia de la clase *AsyncEvent* representa un evento (similar a una interrupción o una señal).

El evento se dispara de dos formas, la primera es ejecutando el método *AsyncEvent.fire*() y la otra es que ocurra un evento en el entorno. Se pueden definir eventos (AsyncEvent) y manejadores (AsyncEventHandler) para los mismos que se ejecutarán cuando los primeros se disparen.

AsyncEvent.addHandler(AsyncEventHandler a)

Pueden asociarse varios manejadores a un mismo evento. Estos se ejecutarán en el orden definido por sus Scheduling y Release parameters. Se garantiza que por cada vez que se dispara el evento habrá

una ejecución de cada uno de los manejadores asociados. Si el manejador tiene asociado SporadicParameters, se asegura que no habrá 2 ejecuciones del mismo en menos del tiempo entre ejecuciones consecutivas indicado.

Pueden definirse zonas en las que tengan efecto los eventos producidos (métodos con throws AsynchronouslyInterruptedException) y zonas en las que no.

4.6.1 CLASS ASYNCEVENT

Dentro de los métodos básicos de esta clase están:

Constructores
AsyncEvent()
AsyncEvent(ReleaseParameters releaseParams) Crea una instancia de AsyncEvent con los parámetros de liberación asociados al evento

Métodos	
void	**addHandler**(AsyncEventHandler handler) Añade el manejador (handler) como uno (puede haber varios) de los asociados a este evento. Cuando se produzca el evento se ejecutará el método run() del dicho manejador
void	**bindTo**(java.lang.String happening) asocia el AsyncEvent a un evento externo. Los valores dependen de la implementación.
ReleaseParameters	**createReleaseParameters**() Crea los parámetros temporales apropiados para el evento AsyncEvent..
void	**fire**() Dispara el evento y entonces provoca la ejecución de los métodos run() de los manejadores asociados.
void	**setHandler**(AsyncEventHandler handler) Asocia un nuevo manejador con este evento, removiendo todos los existentes.
void	**unbindTo**(java.lang.String happening) Desasocia el AsyncEvent del evento externo (happening).

4.6.2 CLASS ASYNCEVENTHANDLER

Implementa las interfaces Schedulable y Runnable. Similares a los threads pero deben ser de mucho menor coste de ejecución. Se ejecutan, por defecto en el thread actual. Si se quiere crear un thread propio para el manejador hay que usar BoundAsyncEventHandler (subclase de AsyncEventHandler).

Constructor
AsyncEventHandler() Crea un manejador que hereda los parámetros (SchedulingParams, ReleaseParam, etc) del thread actual. Existen constructores y métodos en los que se pueden especificar esos parámetros

Métodos	
void	**handleAsyncEvent()** Si el manejador se ha construido usando un objeto que implementa la interface runnable, entonces el método run del objeto es llamado. Este método debe ser invocado repetidamente mientras que `fireCount` sea mayor que cero.
void	**run()** Utilizado por el sistema de captura de eventos asíncronos

Un ejemplo del manejo de interrupciones asíncronas es el siguiente:

```
public class StopableThread extends RealtimeThread{
   public StopableThread (sp s, mp m){};
   public void body() throws AIE{
//código de la tarea todos los métodos deberían lanzar AIE
   }
   public void run() {try{body();}catch (AIE e){};}
}

public class ExternalEvent extends AsyncEvent{
   public void ExternalEvent() {};
   public void native BindTo() {
// código para enganchar un suceso externo a un evento (dependiente de la implementación)
   }
}

public class StopIt extends AsyncEventHandler{
   RealtimeThread t; // thread que vamos a parar
   public StopIt (RealtimeThread T){t = T;}
   public void run() {t.interrupt();}
}

ExternalEvent ee = new ExternalEvent();
ee.BindTo(args); // asociar suceso externo al evento

StopIt Handler = new StopIt; // manejador de evento

StopableThread st = new StopableThread (s,m);
ee.AddHandler (Handler(st));
st.Start();
```

Cuando el suceso externo se produce, este lanza su thread manejador de eventos Handler. Este tiene como función (en este caso) de Interrumpir el thread stel control se pasa al thread st manejador de AIE del thread st

4.7 RELOJES EN JAVA TIEMPO REAL

La clase Clock proporciona el reloj básico en Java tiempo real

Constructor	
Clock()	

Métodos	
AbsoluteTime	**absolute**(RelativeTime time) Devuelve this.getTime() + time.
static Clock	**getRealtimeClock**() Hay siempre un objeto reloj disponible: un reloj en tiempo real que avanza en la sincronización con el mundo externo. Es el reloj básico por defecto.
abstract RelativeTime	**getResolution**() Devuelve la resolución del reloj en tiempo relativo entre ticks.
AbsoluteTime	**getTime**() Devuelve el tiempo actual en un objeto recién asignado.
abstract void	**getTime**(AbsoluteTime time) Devuelve el tiempo actual en un objeto existente.
RelativeTime	**relative**(AbsoluteTime time) Devuelve la diferencia entre AbsoluteTime y el tiempo del reloj.
abstract void	**setResolution**(RelativeTime resolution) Establece la resolución del reloj.

La clase HighResolutionClock proporciona un reloj de alta resolución. La resolución es la misma que la frecuencia a que trabaja el procesador del sistema en el caso de los Pentium

Constructor	
HighResolutionClock()	

Métodos	
RelativeTime	**getResolution**() Devuelve la resolución del reloj en tiempo relativo entre ticks
void	**getTime**(AbsoluteTime time) Devuelve el tiempo actual en un objeto existente de tipo AbsoluteTime
void	**setResolution**(RelativeTime resolution) Establece la resolución del reloj.

La clase RealtimeClock proporciona acceso al tiempo "de Reloj de pared", el reloj de sistema operativo. En la práctica la resolución del reloj es del orden de microsegundos. Su funcionamiento se basa en llamadas a la función del sistema operativo gettimeofday, entonces la resolución que depende de la resolución garantizada por OS subyacente.

Constructor
RealtimeClock()

Métodos	
RelativeTime	**getResolution**() Devuelve la resolución del reloj en tiempo relativo entre ticks.
void	**getTime**(AbsoluteTime time) Devuelve el tiempo actual en un objeto existente de tipo AbsoluteTime .
void	**setResolution**(RelativeTime resolution) Establece la resolución del reloj.

4.8 MEDIDA DEL TIEMPO

jRate implementa el sistema de lectura y de control de tiempo definido en la API de Java tiempo real. Java tiempo real tiene mayor precisión en la medida del tiempo que el el Java clásico. Además permite la definición de tiempos relativos o absolutos. La clase estándar de Java java.util.Date suministra una precisión de milisegundo. En cambio, en aplicaciones de control en tiempo real puede ser insuficiente, requiriendo precisión de nanosegundos. Java para tiempo real implementa un reloj de 64 bits para los milisegundos junto a 32 bits para los nanosegundos dentro del milisegundo.

4.8.1 HIGHRESOLUTIONTIME

Es una clase que permite la expresión del tiempo con una precisión de nanosegundos. Es abstracta y no puede ser usada directamente dado que no tiene constructores públicos. En cambio se puede usar alguna de sus subclases: AbsoluteTime, RelativeTime o RationalTime

Tiene métodos para pasar de tiempos relativos a absolutos (*absolute()*), para comparar tiempos (*compareTo(..), equals(..))* y para obtener el tiempo actual (*set(..)*).

Dentro de sus subclases están:

• AbsoluteTime: Tiempo absoluto desde 1-1-1970 a las 00:00:00 GMT

Constructores
AbsoluteTime() Equivalente a AbsoluteTime(0,0).
AbsoluteTime(AbsoluteTime time) Construye un nuevo objeto desde un objeto AbsoluteTime.
AbsoluteTime(java.util.Date date) Construye un nuevo objeto AbsoluteTime a partir del tiempo en Java clásico
AbsoluteTime(long millis, int nanos) Construye un nuevo objeto AbsoluteTime que sobrepasa en millis milisegundos y nanos nanosegundos las 00:00.00 GMT del 1 de enero de 1970.

Métodos	
AbsoluteTime	**add**(long millis,int nanos) Añade los milisegundos y nanosegundos al objeto dado.
void	**add**(long millis, int nanos, AbsoluteTime dest) Suma al objeto los milisegundos y nanosegundos y el resultado se vuelca en dest
AbsoluteTime	**add**(RelativeTime time) Devuelve this+time
java.util.Date	**getDate**() El tiempo de this en java clásico.
RelativeTime	**relative**(Clock clock) Devuelve el tiempo relativo respecto al reloj dado.
RelativeTime	**subtract**(AbsoluteTime time) Devuelve this - time.

• RelativeTime: Tiempo relativo desde el instante actual.

Constructores
RelativeTime() Equivalente a RelativeTime(0,0).
RelativeTime(long millis, int nanos) Construye un nuevo objeto RelativeTime a partir de los milisegundos y nanosegundos.
RelativeTime(RelativeTime relativeTime) Construct a new RelativeTime object from the given RelativeTime.

Métodos	
AbsoluteTime	**absolute**(Clock clock) Convierte este tiempo en absoluto relativo al reloj dado
RelativeTime	**add**(long millis, int nanos) Suma la cantidad de milisegundos y nanosegundos al tiempo dado .
RelativeTime	**add**(RelativeTime time) Devuelve this + time.

- RationalTime: Subclase de RelativeTime que divide un tiempo en subintervalos de cierta frecuencia. Se utiliza para disparar tareas periódicas.

Constructores
RationalTime(int frequency) Construye un nuevo objeto RationalTime equivalente a RationalTime(1000, 0, frequency). Un objeto que tiene frecuency ocurrencias por segundo
RationalTime(int frequency, long millis, int nanos) Construye un nuevo objeto RationalTime. Un objeto que tiene frecuency ticks en el intervalo dado
RationalTime(int frequency, RelativeTime interval) Construye un nuevo objeto RationalTime. Un objeto que tiene frecuency ticks en el intervalo dado

Métodos	
void	**addInterarrivalTo**(AbsoluteTime destination)

int	**getFrequency**()
RelativeTime	**getInterarrivalTime**() Devuelve el tiempo relativo entre dos ticks.
void	**setFrequency**(int frequency)

Si tengo (3, 100, 50) como RationalTime para un Temporizador periódico, el temporizador se ejecutara 3 veces cada 100ms y 50 nanoseg.

4.9 TEMPORIZADORES

RTSJ permite crear temporizadores periódicos o puntuales a los que se les pueden asociar acciones como a cualquier evento asíncrono. Un Temporizador es una función temporal que mide el tiempo en relación con un Reloj dado. Se utilizará la clase PeriodicTimer para crear eventos que son lanzados repetidamente a intervalos regulares, o OneShotTimer para un evento que se dispara una vez en un tiempo determinado.

4.9.1 TIMER

Es la clase base de los temporizadores, hereda de AsyncEvent. Es una clase abstracta. No se deben definir directamente objetos de la misma

Métodos	
ReleaseParameters	**createReleaseParameters**() Crea el bloque de parámetros apropiados a las características temporales del evento.
void	**destroy**() Para el temporizador y lo devuelve al estado previo al disparo.
void	**disable**() Deshabilita el temporizador.
void	**enable**() Re-Habilita el temporizador
Clock	**getClock**() Devuelve el reloj en que se basa el temporizador
AbsoluteTime	**getFireTime**() Devuelve el tiempo en que el evento disparará
boolean	**isRunning**() Devuelve true si el temporizador esta funcionando.
void	**reschedule**(HighResolutionTime time) Cambia el tiempo de planificación para el evento.

`void`	**`start`**`()` Lanza el temporizador.
`boolean`	**`stop`**`()` Para el temporizador.

Como subclases están:

OneShotTimer

public OneShotTimer (HighResolutionTime time, Clock reloj, AsyncEventHandler manejador)

Define un temporizador que provocará la ejecución del manejador cuando se dispare.

PeriodicTimer

public PeriodicTimer (HighResolutionTime start, RelativeTime intervalo, Clock reloj, AsyncEventHandler manejador)

Define un temporizador que comenzara a contar el tiempo en *start* y se disparara cada *intervalo* provocando la ejecución del manejador.

```
package demo.time;

// -- jTools Import --
import jtools.time.*;
import jtools.jargo.*;

public class TimerDemo {

   private static ArgParser parseArgs(String[] args) throws Exception {
      CommandLineSpec cls = new CommandLineSpec();
      cls.addRequiredArg(new ArgSpec("iteration",
                              new String[]
         {"jtools.jargo.NaturalValidator"}));

      ArgParser argParser = new ArgParser(cls);
      cls = null;
      argParser.parse(args);

      return argParser;
   }

   public static void main(String args[]) throws Exception {
         ArgParser argParser = parseArgs(args);
         ArgValue av = argParser.getArg("iteration");
         int count = ((Integer)av.getValue()).intValue();

         HighResTimer timer = new HighResTimer();
         timer.start();
         for (int i = 0; i < count; i++) { }
         timer.stop();

         System.out.println("Executed " + count
                     + " iteration in: "
                     + timer.getElapsedTime() + " msec");
   }
}
```

4.10 TRANSFERENCIA ASÍNCRONA DEL CONTROL

Java en tiempo real permite cambiar el flujo normal de ejecución de forma asíncrona. Se basa en la extensión de *java.lang.Thread.interrupt*(), en las clases *AsynchronouslyInterruptedException* (AIE), *Timed* y el interface *Interruptible.*

Los requisitos que debe cumplir la Transferencia Asíncrona del Control (ATC) debe cumplir varios principios ya que sino podrían surgir problemas de funcionamiento a la hora de usarlo:

Un Thread debe indicar explícitamente que puede ser objeto de una ATC.

Pueden existir fragmentos de ese Thread protegidos contra ATC.

Una ATC no tiene vuelta al punto en el que se estaba ejecutando cuando se produjo. Si es necesaria la vuelta hay que usar AsyncEvendHandler.

Existirá un mecanismo para disparar de forma explícita un ATC, bien directamente desde otro Thread o indirectamente mediante un AsyncEventHandler

Se podrá disparar una ATC basada en cualquier tipo de evento asíncrono (evento externo, temporizador, disparo desde otra tarea)

Será posible abortar un Thread mediante una ATC sin los riesgos de los métodos stop o destroy de java.lang.Thread.

Si se utiliza el tratamiento de excepciones para una ATC se debe asegurar que una excepción asíncrona sólo se capturará por un manejador específico de la misma y no por uno de propósito general. Se podrán tener Atas anidadas. Si un bloque de código asociado a un temporizador termina antes de que se dispare el temporizador, éste se parará y sus recursos será liberados.

4.10.1 FUNCIONAMIENTO ATC

A la hora de manejar el ATC hay que tener en cuenta que para que un método pueda ser interrumpido debe incluir la cláusula (throws AsynchronousInterruptedException) en su declaración. El mecanismo de ATC entra en funcionamiento si t es una instancia de RealtimeThread o NoHeapRealtimeThread y cualquier tarea del sistema ejecuta *t.interrupt(),* t ha sido asociado a una AIE (con *aie.doInterruptible (t)*) y se ejecuta *aie.fire().*

Cuando se dispara la AIE, si el control se encuentra en una sección no interrumpible (sección sincronizada o un método sin throws AIE), ocurre que se coloca la AIE como pendiente hasta que se vuelva a una sección interrumpible o se invoque a un método interrumpible. Por otro lado si el control se encuentra en una sección interrumpible, ocurre que el control se transfiere a la cláusula catch más próxima que trate esa AIE y no este en una sección interrumpible. Finalmente si el control se encuentra en wait, sleep o join entonces se despierta al thread y se lanza la AIE.

Si el control se transfiere de un método no interrumpible a uno interrumpible mediante la propagación de una excepción y hay una excepción AIE pendiente en el momento de la transición, entonces se descarta la excepción lanzada y se remplaza por la AIE.

4.10.2 ASYNCHRONOUSLYINTERRUPTEDEXCEPTION (AIE)

Esta es una subclase de java.lang.InterruptedException. Es una excepción especial que se lanza en respuesta al intento de realizar una transferencia asíncrona de control en un RealtimeThread. Si un método declara que lanzará (thows) una AIE, ésta se producirá cuando se invoque a RealtimeThread.interrupt() mientras ese método esté ejecutando o, si está pendiente, cuando el control vuelva al mismo. Cuando se captura una interrupción (AIE) se debe invocar el método happened() para comprobar que es la pendiente, con lo que será eliminada del thread, sino se continuará propagando.

Como métodos están:

- *public AsynchronouslyInterruptedException ()*

- *public synchronized boolean disable():* Deshabilita el tratamiento de la interrupción. Si ésta salta mientras está deshabilitada se pone en estado de pendiente y se lanzará en cuanto vuelva a ser habilitada. Sólo debe ser invocado dentro de una llamada a *doInterruptible()*
- *public synchronized boolean enable():* Habilita el tratamiento de la excepción.
- *public boolean doInterruptible (Interruptible tarea):* invoca al método *run()* de la tarea
- *public synchronized boolean fire()* Hace que esa excepción sea la actual si se ha invocado a *doInterruptible()* y no se ha completado
- *public boolean happened (boolean propagar):* Utilizada con una instancia de esa excepción para ver si la actualmente disparada es ella. Propagar indica si ésta se debe propagar o no si no es la actual.

4.10.3 TIMED

Es una subclase de AIE. Se utiliza para realizar una ATC basada en un temporizador (se invocará a *interrupt()* y por tanto se disparará la excepción cuando cumpla el temporizador)

Como métodos están:

- public Timed (HighResolutionTime tiempo): Crea una instancia de Timed que invocará a *interrup()* cuando pase el tiempo indicado después de invocar a *doInterruptible()*
- public boolean doInterruptible(Interruptible tarea): Ejecuta el método run de la tarea y arranca el temporizador

4.10.4 INTERFACE INTERRUPTIBLE

Se utiliza para los objetos que vayan a ser interrumpidos por medio de AIE.doInterruptible();

Como métodos están:

- public void interruptAction (AIE excepcion): Es el método que se ejecuta si el método run es interrumpido. La excepcion indica el AIE generado y sirve para invocar métodos de la misma.
- Public void run (AIE excepcion): Es un método que se ejecuta cuando se invoca a doInterruptible(). El parámetro sirve para lo mismo que en el método anterior.

Un ejemplo de transferencia asíncrona de control seria el siguiente:

```
public class Seleccion implements Interruptible
{
public void Seleccion (args) {
  // constructor
}
public void run() throws AIE {
  // código del algoritmo cada método debería lanzar AIE
}
public void interrupted() {
  // código de respuesta a la interrupción
}
}

Seleccion Sel = new Seleccion(args);
Timed t = new
   Timed(RelativeTime(500,0));
```

```
t.doInterruptible(Sel);

// El método run de Sel terminará de ejecutarse si tarda menos de 500 ms.
// Si tarda más será interrumpido

t.resetTime(RelativeTime(5000,0));
t.doInterruptible(Sel);
// Lo mismo que antes pero con un plazo de 5000 ms.
```

FILÓSOFOS EN TIEMPO REAL

El primer programa que se realiza en Java Real Time será la implementación de la red de los filósofos. Partiendo de la base de la implementación realizada en Java clásico se realizan una serie de modificaciones pero se mantiene la mayor parte del código.

Los hilos de java (Threads) son convertidos a RealtimeThread, Hilos de tiempo real. Los RealtimeThread tienen menor prioridad que el recolector de basura (garbage colector). También se podrían implementar como NoHeapRealtimeThread. Similares a los anteriores pero no coloca objetos en el heap y por tanto puede ejecutar siempre antes que el recolector de basura.

La comunicación entre el hilo coordinador y las transiciones se realiza de forma idéntica que en Java Clásico, mediante monitores y un buffer

5.1 IMPLEMENTACIÓN DE LOS HILOS TRANSICIÓN

```
class transicion implements Runnable{
private int priority;
  private int childNum;
   private RealtimeThread rtThread;
   private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

public transicion(int aidentidad,comunica ac,BoundedBuffer abuffer, int aprioridad){
   identidad=aidentidad;
   bufferproceso = abuffer;
   c =ac;
 this.priority = aprioridad;
```

```
        this.rtThread = new RealtimeThread(this);
        PriorityParameters prioParams = new PriorityParameters(this.priority);
        this.rtThread.setSchedulingParameters(prioParams);

    }

    public void run(){
    aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
    while(true){          //System.out.println (identidad + " esperando autorización") ;
       c.preguntautorizacion();
       //this.yield();
    switch(identidad){
              case TF1_P:System.out.println ("filósofo uno pensando") ;break;
              case TF2_P:System.out.println ("filósofo dos pensando") ;break;
              case TF3_P:System.out.println ("filósofo tres pensando") ;break;
              case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
              case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
              case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
              case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
              case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
              case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
              case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
              }

              for (i = 0; i < 100000000; i++) {
              // comiendo o pensando
              }


       bufferproceso.put(identidad);

     }
     }
    public void start() {
          this.rtThread.start();
       }
    }
```

Como se observa las transiciones no se declaran directamente como hilos de tiempo real, se declaran como una clase que incorpora la interface runnable. Esto es debido a que para construir la transición no es suficiente con el constructor del hilo tiempo real, sino que hay que crear más variables para identificar la transición y permitir la comunicación con el hilo coordinador.

Constructor de la transición

```
    public transicion(int aidentidad,comunica ac,BoundedBuffer abuffer, int aprioridad){
       identidad=aidentidad;
       bufferproceso = abuffer;
       c =ac;
    this.priority = aprioridad;


        this.rtThread = new RealtimeThread(this);
        PriorityParameters prioParams = new PriorityParameters(this.priority);
        this.rtThread.setSchedulingParameters(prioParams);

    }
```

En la linea:

```
this.rtThread = new RealtimeThread(this);
```

Implica que el thread Transición sea construido como un thread de tiempo real, al hacer la conversión mediante la función:

```
RealtimeThread(this);
```

Que como argumento tiene el propio hilo transición.

A continuación se le asigna la prioridad de ejecución con:

```
PriorityParameters prioParams = new PriorityParameters(this.priority);
    this.rtThread.setSchedulingParameters(prioParams);
```

También se debe encapsular la llamada al procedimiento start:

```
public void start() {
    this.rtThread.start();
  }
```

Dado que es un procedimiento que no existe para la clase transición pero si para el hilo de tiempo real que contiene.

5.2 IMPLEMENTACIÓN DEL COORDINADOR

```
class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];

private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
   nombre=anombre;
   vectorcomunica=avectorcomunica;
   M=aestado;
   buffercoor=abuffercoor;
this.rtThread = new RealtimeThread(this);
    PriorityParameters prioParams = new PriorityParameters(22);
    this.rtThread.setSchedulingParameters(prioParams);
 }

public void run(){
System.out.println (nombre + " empieza ejecución");

while(true){

        if (M[P3]) {
        System.out.println (  "filósofo tres autorizado a pensar") ;

        M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
        }
```

```
if (M[P1]) {
System.out.println (  "filósofo uno autorizado a pensar") ;

M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
}

if (M[P2]) {
System.out.println (  "filósofo dos autorizado a pensar") ;

M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
}

if (M[P4]) {
System.out.println (  "filósofo cuatro autorizado a pensar") ;

M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
}

if (M[P5]) {
System.out.println (  "filósofo cinco autorizado a pensar") ;

M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
}
if          (M[T2] && M[T3] && M[C3]){
System.out.println (  "filósofo tres autorizado a comer") ;

M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
}

if          (M[T5] && M[T1] && M[C1]){
System.out.println (  "filósofo uno autorizado a comer") ;

M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
}

if          (M[T1] && M[T2] && M[C2]){
System.out.println (  "filósofo dos autorizado a comer") ;

M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
}



if          (M[T3] && M[T4] && M[C4]){
System.out.println (  "filósofo cuatro autorizado a comer") ;

M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
}

if          (M[T4] && M[T5] && M[C5]){
System.out.println (  "filósofo cinco autorizado a comer") ;

M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
}

if (buffercoor.numberInBuffer>0)
```

```
   { System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
         //impresión de la red de petri
   for (i = 0; i < 15; i++) {
         System.out.print (  M[i]+ " ");
         }
   System.out.println ( );
         }
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????

while(buffercoor.numberInBuffer>0) {
         valor= buffercoor.get();
   switch(valor){
         case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
         case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
         case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
         case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
         case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
         case  TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de
comer");break;
         case  TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de
comer");break;
         case  TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de
comer");break;
         case  TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println (  "filósofo cuatro termina de
comer");break;
         case  TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println (  "filósofo cinco termina de
comer");break;
         }// end swicth


   }//end while


   }//end while true
        } //en run
public void start() {
      this.rtThread.start();
   }
}//
```

Como se observa los cambios realizados respecto al programa en Java clásico son los mismos que para los hilos transición.

5.3 PROGRAMA

```
import javax.realtime.*;
class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}

class comunica {// monitores utilizados por el coordinador para autorizar la ejecución
// de transciones

private boolean disponible = false;
```

```
public synchronized void preguntautorizacion()
        {
         while(!disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
                 disponible=false;
                 //notify();

        }
public synchronized void autorizoejecucion()
        {
                //while(disponible){
                        //try
                //      {
//                                      wait();
                        //}
                    //  catch(InterruptedException e){}
                   //}
                        disponible=true;
                        notify();
        }
}

class transicion implements Runnable{
private int priority;
  private int childNum;
   private RealtimeThread rtThread;
   private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

public transicion(int aidentidad,comunica ac,BoundedBuffer abuffer, int aprioridad){
          identidad=aidentidad;
          bufferproceso = abuffer;
          c =ac;
 this.priority = aprioridad;


       this.rtThread = new RealtimeThread(this);
       PriorityParameters prioParams = new PriorityParameters(this.priority);
       this.rtThread.setSchedulingParameters(prioParams);

        }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){//System.out.println (identidad + " esperando autorización") ;
          c.preguntautorizacion();
          //this.yield();
switch(identidad){
                case TF1_P:System.out.println ("filósofo uno pensando") ;break;
```

```
            case TF2_P:System.out.println ("filósofo dos pensando") ;break;
            case TF3_P:System.out.println ("filósofo tres pensando") ;break;
            case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
            case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
            case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
            case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
            case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
            case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
            case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
                }

            for (i = 0; i < 100000000; i++) {
            // comiendo o pensando
                }


      bufferproceso.put(identidad);

     }
     }
 public void start() {
     this.rtThread.start();
  }
}

class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];

private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
      nombre=anombre;
      vectorcomunica=avectorcomunica;
      M=aestado;
      buffercoor=abuffercoor;
this.rtThread = new RealtimeThread(this);
    PriorityParameters prioParams = new PriorityParameters(22);
    this.rtThread.setSchedulingParameters(prioParams);
     }

public void run(){
System.out.println (nombre + " empieza ejecución");

while(true){

            if (M[P3]) {
            System.out.println (  "filósofo tres autorizado a pensar") ;

            M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
```

```
    }

if (M[P1]) {
System.out.println (  "filósofo uno autorizado a pensar") ;

M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
    }

if (M[P2]) {
System.out.println (  "filósofo dos autorizado a pensar") ;

M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
    }

if (M[P4]) {
System.out.println (  "filósofo cuatro autorizado a pensar") ;

M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
    }

if (M[P5]) {
System.out.println (  "filósofo cinco autorizado a pensar") ;

M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
    }
    if   (M[T2] && M[T3] && M[C3]){
System.out.println (  "filósofo tres autorizado a comer") ;

M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
    }

    if   (M[T5] && M[T1] && M[C1]){
System.out.println (  "filósofo uno autorizado a comer") ;

M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
    }

    if   (M[T1] && M[T2] && M[C2]){
System.out.println (  "filósofo dos autorizado a comer") ;

M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
    }



    if   (M[T3] && M[T4] && M[C4]){
System.out.println (  "filósofo cuatro autorizado a comer") ;

M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
    }

    if   (M[T4] && M[T5] && M[C5]){
System.out.println (  "filósofo cinco autorizado a comer") ;

M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
    }
```

```
            if (buffercoor.numberInBuffer>0)
        { System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
            //impresión de la red de petri
        for (i = 0; i < 15; i++) {
            System.out.print (  M[i]+ " ");
                }
        System.out.println ( );
                }
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????

while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
        switch(valor){
            case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
            case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
            case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
            case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
            case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
            case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
            case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
            case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
            case TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println (  "filósofo cuatro termina de
comer");break;
            case TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println (  "filósofo cinco termina de
comer");break;
            }// end swicth


        }//end while


        }//end while true
        } //en run
public void start() {
      this.rtThread.start();
   }
}// end coordinador

public class rtfilguardas{



   public static void main(String[] args) {
      //RealTimeRunner runner = new RealTimeRunner(10, 5);
     // runner.start();

   int i;

// definición de contantes para las transiciones
int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

// definición de contantes para los lugares
int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;

// estado incial de la red de petri
boolean[] estado={true,false,true,true,false,true,true,false,true,true,false,true,true,false,true};
```

```
// vector de monitores para comunicar la acepatación del comienzo del disparo de las transiciones

System.out.println (  "hola") ;
comunica com1=new comunica();
comunica com2=new comunica();
comunica com1p=new comunica();
comunica com1c=new comunica();
comunica com2p=new comunica();
comunica com2c=new comunica();
comunica com3p=new comunica();
comunica com3c=new comunica();
comunica com4p=new comunica();
comunica com4c=new comunica();
comunica com5p=new comunica();
comunica com5c=new comunica();

comunica[] comunicaarray = {com1p,com1c,com2p,com2c,com3p,com3c,com4p,com4c,com5p,com5c};

System.out.println (  "hola1") ;

// buffer para comunicar los fin de code
BoundedBuffer buffer= new BoundedBuffer(10);

System.out.println (  "hola2") ;

transicion TF1_Ptask= new transicion(TF1_P,comunicaarray[0], buffer,11);
transicion TF1_Ctask= new transicion(TF1_C,comunicaarray[1], buffer,12);
transicion TF2_Ptask= new transicion(TF2_P,comunicaarray[2], buffer,13);
transicion TF2_Ctask= new transicion(TF2_C,comunicaarray[3], buffer,14);
transicion TF3_Ptask= new transicion(TF3_P,comunicaarray[4], buffer,15);
transicion TF3_Ctask= new transicion(TF3_C,comunicaarray[5], buffer,16);
transicion TF4_Ptask= new transicion(TF4_P,comunicaarray[6], buffer,17);
transicion TF4_Ctask= new transicion(TF4_C,comunicaarray[7], buffer,18);
transicion TF5_Ptask= new transicion(TF5_P,comunicaarray[8], buffer,19);
transicion TF5_Ctask= new transicion(TF5_C,comunicaarray[9], buffer,20);

transicion[]                                                                    Transicionarray=
{TF1_Ptask,TF1_Ctask,TF2_Ptask,TF2_Ctask,TF3_Ptask,TF3_Ctask,TF4_Ptask,TF4_Ctask,TF5_Ptask,TF5_Ctask};

coordinador coor=new coordinador("COORDINADOR FILÓSOFOS",comunicaarray,estado,buffer);
coor.start();
System.out.println (  "hola3") ;


for (i = 0; i < 10; i++) {

              Transicionarray[i].start();
            }
System.out.println (  "hola4") ;




}
}
```

Con el programa anterior al ejecutarse en tiempo real la máquina Linux dejaba de responder, esto era debido a que se lanzaba un hilo, el coordinador, con prioridad alta, este no se suspendía nunca, impidiendo la ejecución de las tareas de sistema de Linux. Por lo tanto la tarea no se podía ni cancelar.

En cambio si se ejecuta como usuario normal, el programa se ejecutaba correctamente debido a que las prioridades en java no son estrictas, sino que a más prioridad, más tiempo de CPU le toca al hilo, "garantizando" que todas las tareas se van a ejecutar.

Ejemplo de ejecución en clásico;

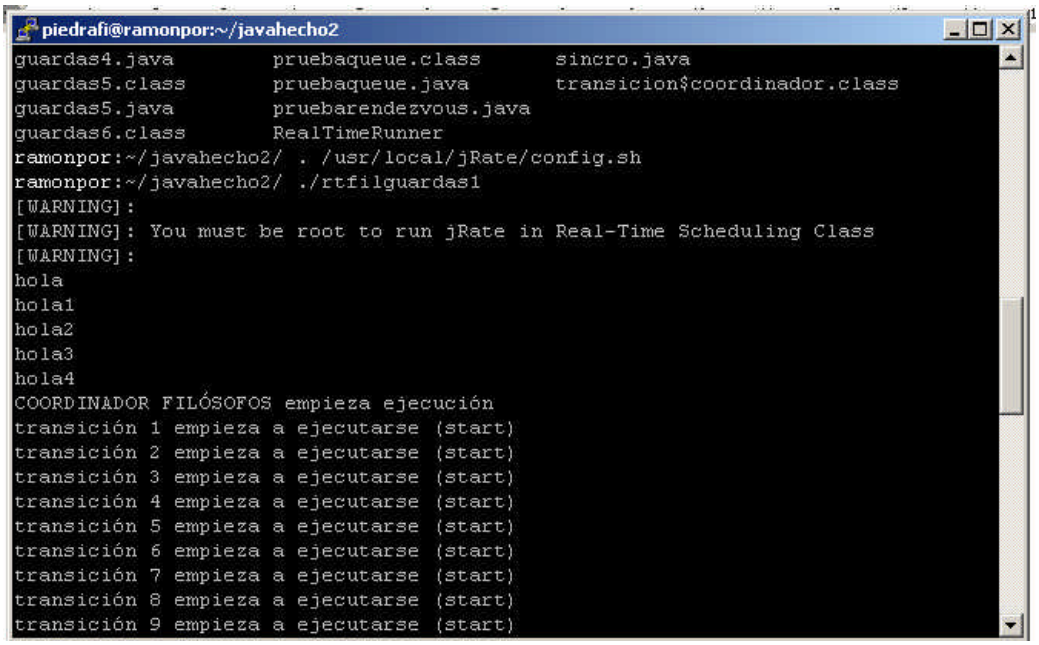

Figura 7

Ejemplo de cuelgue (ejecución en tiempo real):

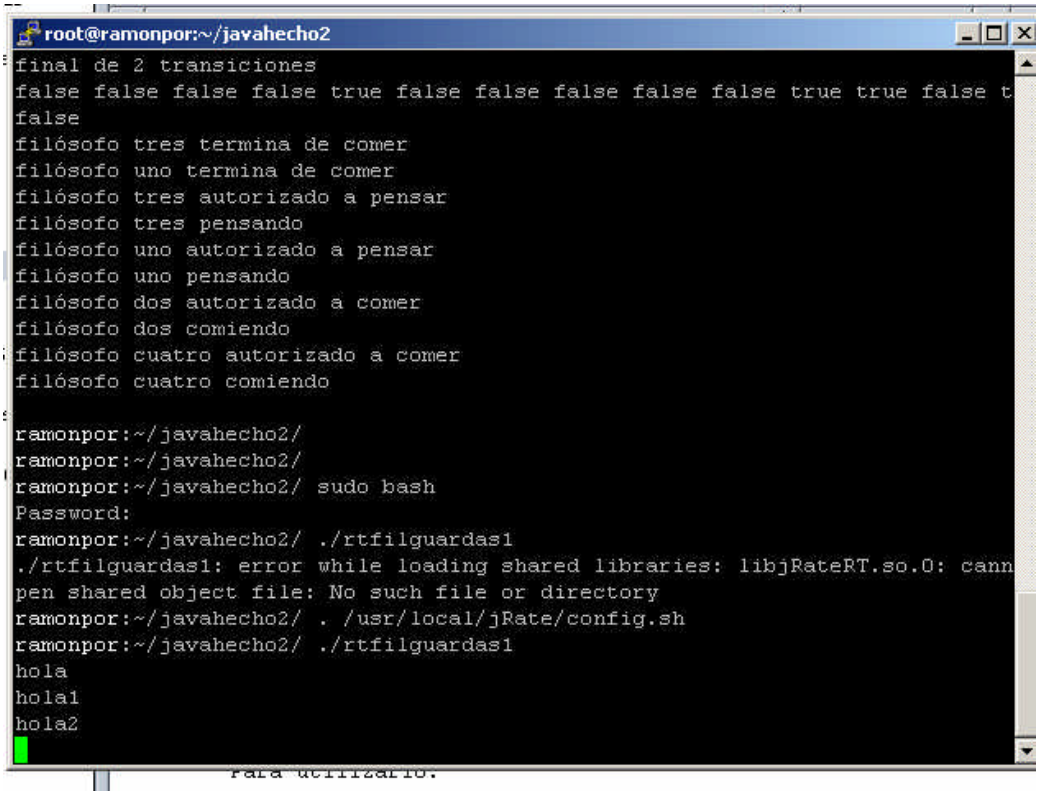
root@ramonpor:~/javahecho2
final de 2 transiciones
false false false false true false false false false false true true false t
false
filósofo tres termina de comer
filósofo uno termina de comer
filósofo tres autorizado a pensar
filósofo tres pensando
filósofo uno autorizado a pensar
filósofo uno pensando
filósofo dos autorizado a comer
filósofo dos comiendo
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
ramonpor:~/javahecho2/
ramonpor:~/javahecho2/
ramonpor:~/javahecho2/ sudo bash
Password:
ramonpor:~/javahecho2/ ./rtfilguardas1
./rtfilguardas1: error while loading shared libraries: libjRateRT.so.0: cann
pen shared object file: No such file or directory
ramonpor:~/javahecho2/ . /usr/local/jRate/config.sh
ramonpor:~/javahecho2/ ./rtfilguardas1
hola
hola1
hola2

Figura 8

5.4 SUSPENSIÓN DEL COORDINADOR

```
class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];

private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
        nombre=anombre;
        vectorcomunica=avectorcomunica;
        M=aestado;
        buffercoor=abuffercoor;
        this.rtThread = new RealtimeThread(this);
      PriorityParameters prioParams = new PriorityParameters(50);
      this.rtThread.setSchedulingParameters(prioParams);
            }
//public void sleep() { }
public void run(){
System.out.println (nombre + " empieza ejecución");
while(true){
//for (j = 0; j < 500000000; j++) {


            if (M[P3]) {
            System.out.println (  "filósofo tres autorizado a pensar") ;

            M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
                }

            if (M[P1]) {
            System.out.println (  "filósofo uno autorizado a pensar") ;

            M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
                }

            if (M[P2]) {
            System.out.println (  "filósofo dos autorizado a pensar") ;

            M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
                }

            if (M[P4]) {
            System.out.println (  "filósofo cuatro autorizado a pensar") ;

            M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
                }

            if (M[P5]) {
```

```
            System.out.println (  "filósofo cinco autorizado a pensar") ;

            M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
                }
                if   (M[T2] && M[T3] && M[C3]){
            System.out.println (  "filósofo tres autorizado a comer") ;

            M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
                }

                if   (M[T5] && M[T1] && M[C1]){
            System.out.println (  "filósofo uno autorizado a comer") ;

            M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
                }

                if   (M[T1] && M[T2] && M[C2]){
            System.out.println (  "filósofo dos autorizado a comer") ;

            M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
                }



                if   (M[T3] && M[T4] && M[C4]){
            System.out.println (  "filósofo cuatro autorizado a comer") ;

            M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
                }

                if   (M[T4] && M[T5] && M[C5]){
            System.out.println (  "filósofo cinco autorizado a comer") ;

            M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
                }

            if (buffercoor.numberInBuffer>0)
        { System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
            //impresión de la red de petri
        for (i = 0; i < 15; i++) {
            System.out.print (  M[i]+ " ");
                }
        System.out.println ( );
                }
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????


while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
        switch(valor){
            case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
            case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
            case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
            case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
            case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
            case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
            case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
```

```
            case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println ( "filósofo tres termina de comer");break;
            case TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println ( "filósofo cuatro termina de comer");break;
            case TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println ( "filósofo cinco termina de comer");break;
            }// end swicth

        }//end while

          try
                {
                        this.rtThread.sleep(1000);
                }
                catch(InterruptedException e){}

        }//end while true
        } //en run
public void start() {
      this.rtThread.start();
   }
}// end coordinador
```

Con este código:

```
  try
                {
                        this.rtThread.sleep(1000);
                }
                catch(InterruptedException e){}
```

Hacemos que el coordinador se duerma durante un segundo dando tiempo a ejecutar las demás tareas

5.5 EJECUCIÓN PERIÓDICA DEL COORDINADOR

```
class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];
 boolean retVal;



private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
        nombre=anombre;
        vectorcomunica=avectorcomunica;
        M=aestado;
```

```
        buffercoor=abuffercoor;
        this.rtThread = new RealtimeThread(this);
      PriorityParameters prioParams = new PriorityParameters(50);
      this.rtThread.setSchedulingParameters(prioParams);
        PeriodicParameters perioParams= new PeriodicParameters(null, new RelativeTime(100,0),new
RelativeTime(30,0),new RelativeTime(50,0),null,null);
        this.rtThread.setReleaseParameters(perioParams);

             }
//public void sleep() { }
public void run(){
System.out.println (nombre + " empieza ejecución");
while(true){
//for (j = 0; j < 500000000; j++) {


            if (M[P3]) {
            System.out.println (  "filósofo tres autorizado a pensar") ;

            M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
                }

            if (M[P1]) {
            System.out.println (  "filósofo uno autorizado a pensar") ;

            M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
                }

            if (M[P2]) {
            System.out.println (  "filósofo dos autorizado a pensar") ;

            M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
                }

            if (M[P4]) {
            System.out.println (  "filósofo cuatro autorizado a pensar") ;

            M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
                }

            if (M[P5]) {
            System.out.println (  "filósofo cinco autorizado a pensar") ;

            M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
                }
               if   (M[T2] && M[T3] && M[C3]){
            System.out.println (  "filósofo tres autorizado a comer") ;

            M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
                }

               if   (M[T5] && M[T1] && M[C1]){
            System.out.println (  "filósofo uno autorizado a comer") ;

            M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
                }
```

```
            if   (M[T1] && M[T2] && M[C2]){
        System.out.println (  "filósofo dos autorizado a comer") ;

        M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
            }



            if   (M[T3] && M[T4] && M[C4]){
        System.out.println (  "filósofo cuatro autorizado a comer") ;

        M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
            }

            if   (M[T4] && M[T5] && M[C5]){
        System.out.println (  "filósofo cinco autorizado a comer") ;

        M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
            }

            if (buffercoor.numberInBuffer>0)
    { System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
            //impresión de la red de petri
    for (i = 0; i < 15; i++) {
            System.out.print (  M[i]+ " ");
                }
    System.out.println ( );
                }
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????


while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
    switch(valor){
            case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
            case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
            case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
            case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
            case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
            case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
            case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
            case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
            case   TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println   (   "filósofo   cuatro   termina   de
comer");break;
            case   TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println   (   "filósofo   cinco   termina   de
comer");break;
            }// end swicth

      //if(buffercoor.numberInBuffer>0)valor= buffercoor.get();
        }//end while
  retVal = this.rtThread.waitForNextPeriod();

        }//end while true
       } //en run
public void start() {
     this.rtThread.start();
   }
```

}//

Hemos hecho que el coordinador se ejecute de forma periódica.

En el constructor:

```
PeriodicParameters perioParams= new PeriodicParameters(null, new RelativeTime(100,0),new RelativeTime(30,0),new RelativeTime(50,0),null,null);
  this.rtThread.setReleaseParameters(perioParams);
```

En el código de ejecución:

```
retVal = this.rtThread.waitForNextPeriod();
```

Con esta instrucción el coordinador se duerme hasta el siguiente periodo

No obstante si se ejecuta el programa, el mismo se aborta, eso es debido a que la instrucción se encuentra al final del código periódico y en la primera ejecución a la máquina virtual java no le da tiempo a cargar y ejecutar el código.

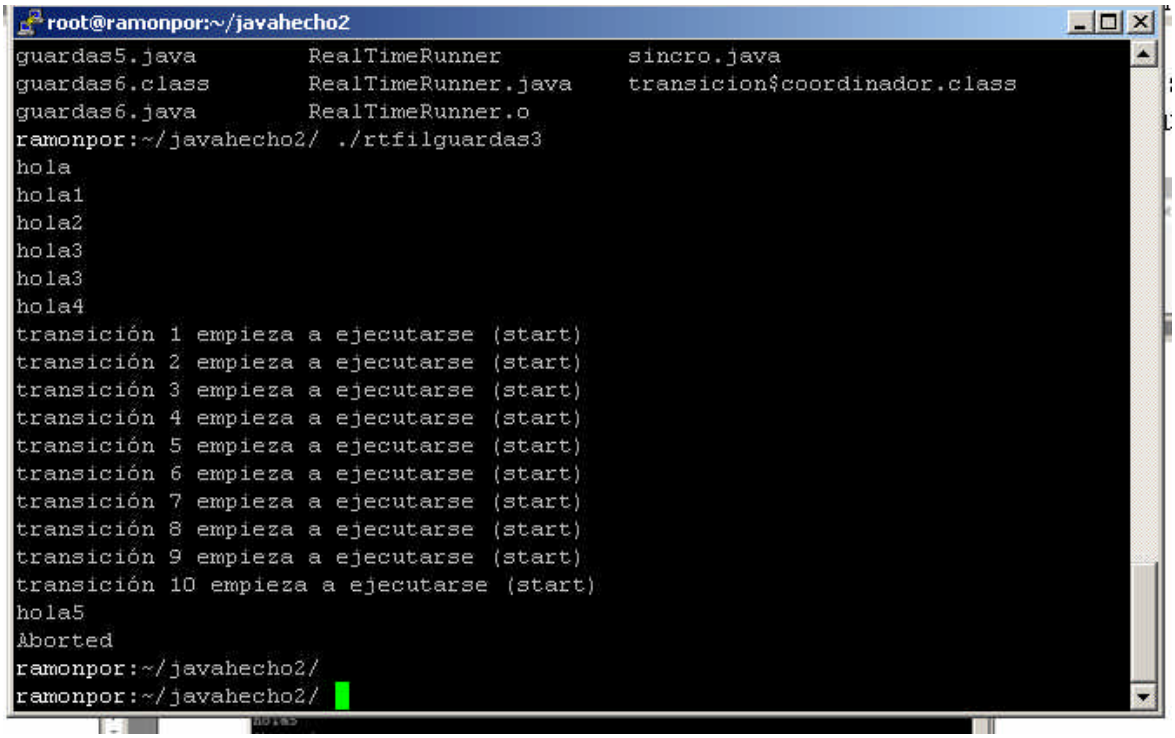

Figura 9

Como mejora situamos el código

retVal = this.rtThread.waitForNextPeriod();

A principio del código periódico

Por lo tanto la ejecución del coordinador es:

Esperar hasta el próximo periodo

Test de sensibilización

Fin de code

```
import javax.realtime.*;


class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}

class comunica {// monitores utilizados por el coordinador para autorizar la ejecución
// de transciones

private boolean disponible = false;

public synchronized void preguntautorizacion()
```

```
        {
         while(!disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
                 disponible=false;
                 //notify();

        }
public synchronized void autorizoejecucion()
        {
                 //while(disponible){
                        //try
                //        {
//                                wait();
                        //}
                    //  catch(InterruptedException e){}
                   //}
                        disponible=true;
                        notify();
        }
}

class transicion implements Runnable{
private int priority;
  private int childNum;
   private RealtimeThread rtThread;
   private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

public transicion(int aidentidad,comunica ac,BoundedBuffer abuffer, int aprioridad){
         identidad=aidentidad;
         bufferproceso = abuffer;
         c =ac;
         this.priority = aprioridad;
         this.rtThread = new RealtimeThread(this);
      PriorityParameters prioParams = new PriorityParameters(this.priority);
      this.rtThread.setSchedulingParameters(prioParams);
                }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){//System.out.println (identidad + " esperando autorización") ;
         c.preguntautorizacion();
         //this.yield();
switch(identidad){
                case TF1_P:System.out.println ("filósofo uno pensando") ;break;
                case TF2_P:System.out.println ("filósofo dos pensando") ;break;
                case TF3_P:System.out.println ("filósofo tres pensando") ;break;
                case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
                case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
```

```
            case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
            case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
            case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
            case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
            case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
                }

            for (i = 0; i < 1000; i++) {
            // comiendo o pensando
                }


        bufferproceso.put(identidad);

       }
       }
 public void start() {
      this.rtThread.start();
   }
}

class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];
 boolean retVal;



private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
        nombre=anombre;
        vectorcomunica=avectorcomunica;
        M=aestado;
        buffercoor=abuffercoor;
        this.rtThread = new RealtimeThread(this);
      PriorityParameters prioParams = new PriorityParameters(50);
      this.rtThread.setSchedulingParameters(prioParams);
        PeriodicParameters perioParams= new PeriodicParameters(null, new RelativeTime(100,0),new
RelativeTime(30,0),new RelativeTime(50,0),null,null);
        this.rtThread.setReleaseParameters(perioParams);

            }
//public void sleep() { }
public void run(){
System.out.println (nombre + " empieza ejecución");
while(true){
 retVal = this.rtThread.waitForNextPeriod();


            if (M[P3]) {
```

```
System.out.println (  "filósofo tres autorizado a pensar") ;

M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
    }

if (M[P1]) {
System.out.println (  "filósofo uno autorizado a pensar") ;

M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
    }

if (M[P2]) {
System.out.println (  "filósofo dos autorizado a pensar") ;

M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
    }

if (M[P4]) {
System.out.println (  "filósofo cuatro autorizado a pensar") ;

M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
    }

if (M[P5]) {
System.out.println (  "filósofo cinco autorizado a pensar") ;

M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
    }
    if   (M[T2] && M[T3] && M[C3]){
System.out.println (  "filósofo tres autorizado a comer") ;

M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
    }

    if   (M[T5] && M[T1] && M[C1]){
System.out.println (  "filósofo uno autorizado a comer") ;

M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
    }

    if   (M[T1] && M[T2] && M[C2]){
System.out.println (  "filósofo dos autorizado a comer") ;

M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
    }



    if   (M[T3] && M[T4] && M[C4]){
System.out.println (  "filósofo cuatro autorizado a comer") ;

M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
    }

    if   (M[T4] && M[T5] && M[C5]){
System.out.println (  "filósofo cinco autorizado a comer") ;
```

```
            M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
                }

            if (buffercoor.numberInBuffer>0)
        { System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
            //impresión de la red de petri
        for (i = 0; i < 15; i++) {
            System.out.print (  M[i]+ " ");
                }
        System.out.println ( );
                }
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????


while(buffercoor.numberInBuffer>0) {
            valor= buffercoor.get();
        switch(valor){
            case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
            case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
            case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
            case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
            case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
            case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
            case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
            case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
            case  TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println  (  "filósofo  cuatro  termina  de
comer");break;
            case  TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println  (  "filósofo  cinco  termina  de
comer");break;
            }// end swicth

        }//end while
          //try

        }//end while true
        } //en run
public void start() {
      this.rtThread.start();
   }
}// end coordinador

class rtfilguardas{



   public static void main(String[] args) {
      //RealTimeRunner runner = new RealTimeRunner(10, 5);
     // runner.start();

   int i;

// definición de contantes para las transiciones
int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

// definición de contantes para los lugares
int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
```

```
// estado incial de la red de petri
boolean[] estado={true,false,true,true,false,true,true,false,true,true,false,true,true,false,true};

// vector de monitores para comunicar la acepatación del comienzo del disparo de las transiciones

System.out.println (  "hola") ;
comunica com1=new comunica();
comunica com2=new comunica();
comunica com1p=new comunica();
comunica com1c=new comunica();
comunica com2p=new comunica();
comunica com2c=new comunica();
comunica com3p=new comunica();
comunica com3c=new comunica();
comunica com4p=new comunica();
comunica com4c=new comunica();
comunica com5p=new comunica();
comunica com5c=new comunica();

comunica[] comunicaarray = {com1p,com1c,com2p,com2c,com3p,com3c,com4p,com4c,com5p,com5c};

System.out.println (  "hola1") ;

// buffer para comunicar los fin de code
BoundedBuffer buffer= new BoundedBuffer(10);

System.out.println (  "hola2") ;

transicion TF1_Ptask= new transicion(TF1_P,comunicaarray[0], buffer,15);
System.out.println (  "hola3") ;

transicion TF1_Ctask= new transicion(TF1_C,comunicaarray[1], buffer,16);
transicion TF2_Ptask= new transicion(TF2_P,comunicaarray[2], buffer,17);
transicion TF2_Ctask= new transicion(TF2_C,comunicaarray[3], buffer,18);
transicion TF3_Ptask= new transicion(TF3_P,comunicaarray[4], buffer,19);
transicion TF3_Ctask= new transicion(TF3_C,comunicaarray[5], buffer,20);
transicion TF4_Ptask= new transicion(TF4_P,comunicaarray[6], buffer,21);
transicion TF4_Ctask= new transicion(TF4_C,comunicaarray[7], buffer,22);
transicion TF5_Ptask= new transicion(TF5_P,comunicaarray[8], buffer,23);
transicion TF5_Ctask= new transicion(TF5_C,comunicaarray[9], buffer,24);
System.out.println (  "hola3") ;

transicion[]                                                                    Transicionarray=
{TF1_Ptask,TF1_Ctask,TF2_Ptask,TF2_Ctask,TF3_Ptask,TF3_Ctask,TF4_Ptask,TF4_Ctask,TF5_Ptask,TF5_Ctask};
coordinador coor=new coordinador("COORDINADOR FILÓSOFOS",comunicaarray,estado,buffer);


System.out.println (  "hola4") ;


for (i = 0; i < 10; i++) {

              Transicionarray[i].start();
             }
System.out.println (  "hola5") ;
coor.start();
```

}
}

Ejecución de rtfilguardas3

```
ramonpor:~/javahecho2/ sudo bash
Password:
ramonpor:~/javahecho2/ . /usr/local/jRate/config.sh
ramonpor:~/javahecho2/ ./rtfilguardas
hola
hola1
hola2
hola3
hola3
hola4
transición 3 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 1 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
hola5
hola6
COORDINADOR FILÓSOFOS empieza ejecución
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro pensando
filósofo tres pensando
filósofo dos pensando
filósofo uno pensando
final de 5 transiciones
false false true false false true false false true false false true false false
true
filósofo cinco termina de pensar
filósofo cuatro termina de pensar
filósofo tres termina de pensar
filósofo dos termina de pensar
filósofo uno termina de pensar
```

filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo tres comiendo
filósofo uno comiendo
final de 2 transiciones
false false false false true false false false false false true true false true
false
filósofo tres termina de comer
filósofo uno termina de comer
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
false false false false false false false false false false false false tr
ue true
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
false false false false false false false false false false true false tru
e false
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
false false false false false false false false false false false false tr
ue true
filósofo cuatro termina de comer
filósofo tres termina de pensar

```
filósofo dos termina de comer
filósofo uno termina de pensar
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
false false false false false false false false false false true false tru
e false
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
```

Los disparos de las transiciones se producen al final del código secuencial (fin de code), y los test de habilitación con las transiciones disparadas deben esperar un periodo. No es muy correcto.

Una mejora es adoptar la siguiente estructura

Esperar hasta el próximo periodo

Fin de code

Test de sensibilización

```
import javax.realtime.*;


class BoundedBuffer  {// buffer para los fín de code
  private int buffer[];
  private int first;
  private int last;
  int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  }
public synchronized void put(int item)

  {
     while(numberInBuffer == size){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     last = (last + 1) % size ; // % modulo
     numberInBuffer++;
     buffer[last] = item;
     notify();
  }

  public synchronized int get()
  {
     while(numberInBuffer == 0){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }

     first = (first + 1) % size ; // % modulo
     numberInBuffer--;
     notify();
     return buffer[first];
  }
}

class comunica {// monitores utilizados por el coordinador para autorizar la ejecución
// de transciones

private boolean disponible = false;

public synchronized void preguntautorizacion()
```

```
        {
         while(!disponible){
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e){}
                }
                 disponible=false;
                 //notify();

        }
public synchronized void autorizoejecucion()
        {
                 //while(disponible){
                        //try
                //      {
//                                wait();
                        //}
                    //  catch(InterruptedException e){}
                   //}
                        disponible=true;
                        notify();
        }
}

class transicion implements Runnable{
private int priority;
  private int childNum;
   private RealtimeThread rtThread;
   private comunica c;
private BoundedBuffer bufferproceso;
private int i;int identidad;int aux;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

public transicion(int aidentidad,comunica ac,BoundedBuffer abuffer, int aprioridad){
         identidad=aidentidad;
         bufferproceso = abuffer;
         c =ac;
         this.priority = aprioridad;
         this.rtThread = new RealtimeThread(this);
      PriorityParameters prioParams = new PriorityParameters(this.priority);
      this.rtThread.setSchedulingParameters(prioParams);
                }

public void run(){
aux = identidad+1;System.out.println ("transición "+aux+ " empieza a ejecutarse (start)") ;
while(true){//System.out.println (identidad + " esperando autorización") ;
         c.preguntautorizacion();
         //this.yield();
switch(identidad){
                case TF1_P:System.out.println ("filósofo uno pensando") ;break;
                case TF2_P:System.out.println ("filósofo dos pensando") ;break;
                case TF3_P:System.out.println ("filósofo tres pensando") ;break;
                case TF4_P:System.out.println ("filósofo cuatro pensando") ;break;
                case TF5_P:System.out.println ("filósofo cinco pensando") ;break;
```

```
                case TF1_C:System.out.println ("filósofo uno comiendo") ;break;
                case TF2_C:System.out.println ("filósofo dos comiendo") ;break;
                case TF3_C:System.out.println ("filósofo tres comiendo") ;break;
                case TF4_C:System.out.println ("filósofo cuatro comiendo") ;break;
                case TF5_C:System.out.println ("filósofo cinco comiendo") ;break;
                    }

                for (i = 0; i < 10000000; i++) {
                // comiendo o pensando
                    }


          bufferproceso.put(identidad);

         }
         }
 public void start() {
       this.rtThread.start();
    }
}

class coordinador implements Runnable{
private RealtimeThread rtThread;
private String nombre;
final int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;
final int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;
private int i,j,valor;
boolean[] M;
comunica vectorcomunica[];
 boolean retVal;



private BoundedBuffer buffercoor;
// al cooordinador hay que darle todo

coordinador(String anombre,comunica[] avectorcomunica,boolean[] aestado,BoundedBuffer abuffercoor){
          nombre=anombre;
          vectorcomunica=avectorcomunica;
          M=aestado;
          buffercoor=abuffercoor;
          this.rtThread = new RealtimeThread(this);
       PriorityParameters prioParams = new PriorityParameters(25);
       this.rtThread.setSchedulingParameters(prioParams);
          PeriodicParameters perioParams= new PeriodicParameters( new RelativeTime(0,0), new RelativeTime(500,0),new
RelativeTime(500,0),new RelativeTime(500,0),null,null);
          this.rtThread.setReleaseParameters(perioParams);

               }
//public void sleep() { }
public void run(){
System.out.println (nombre + " empieza ejecución");
while(true){


//se ejecuta primero la espera para que en la primera ejecución este disponible todo el periodo
                 retVal = this.rtThread.waitForNextPeriod();
```

```
// puede haber problemas de concurrencia aqui si terminan transiciones mientras estamos ejecutando el while?????


// ejecutamos los fin de code
if (buffercoor.numberInBuffer>0)
        System.out.println (  "final de "+buffercoor.numberInBuffer+" transiciones" );
while(buffercoor.numberInBuffer>0) {
        valor= buffercoor.get();
    switch(valor){
        case TF1_P:M[C1]=true;System.out.println (  "filósofo uno termina de pensar") ;break;
        case TF2_P:M[C2]=true;System.out.println (  "filósofo dos termina de pensar") ;break;
        case TF3_P:M[C3]=true;System.out.println (  "filósofo tres termina de pensar") ;break;
        case TF4_P:M[C4]=true;System.out.println (  "filósofo cuatro termina de pensar") ;break;
        case TF5_P:M[C5]=true;System.out.println (  "filósofo cinco termina de pensar") ;break;
        case TF1_C:M[T5]=true;M[T1]=true;M[P1]=true;System.out.println (  "filósofo uno termina de comer");break;
        case TF2_C:M[T1]=true;M[T2]=true;M[P2]=true;System.out.println (  "filósofo dos termina de comer");break;
        case TF3_C:M[T2]=true;M[T3]=true;M[P3]=true;System.out.println (  "filósofo tres termina de comer");break;
        case TF4_C:M[T3]=true;M[T4]=true;M[P4]=true;System.out.println (  "filósofo cuatro termina de
comer");break;
        case TF5_C:M[T4]=true;M[T5]=true;M[P5]=true;System.out.println (  "filósofo cinco termina de
comer");break;

    }// end swicth
        if (buffercoor.numberInBuffer==0)
            {
        //impresión de la red de petri
        System.out.println ( " F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5 ");
        for (i = 0; i < 15; i++) {
        System.out.print (  M[i]+ " ");
            }
        System.out.println ( );
            }
    }//end while


//test de sensibilización
if (M[P3]) {
        System.out.println (  "filósofo tres autorizado a pensar") ;

        M[P3]= false; vectorcomunica[TF3_P].autorizoejecucion();
            }

        if (M[P1]) {
        System.out.println (  "filósofo uno autorizado a pensar") ;

        M[P1]= false; vectorcomunica[TF1_P].autorizoejecucion();
            }

        if (M[P2]) {
        System.out.println (  "filósofo dos autorizado a pensar") ;

        M[P2]= false;vectorcomunica[TF2_P].autorizoejecucion();
            }

        if (M[P4]) {
        System.out.println (  "filósofo cuatro autorizado a pensar") ;
```

```
            M[P4]= false;vectorcomunica[TF4_P].autorizoejecucion();
                }

            if (M[P5]) {
            System.out.println (  "filósofo cinco autorizado a pensar") ;

            M[P5]= false;vectorcomunica[TF5_P].autorizoejecucion();
                }
                if   (M[T2] && M[T3] && M[C3]){
            System.out.println (  "filósofo tres autorizado a comer") ;

            M[T2]= false;M[T3]= false;M[C3]= false;vectorcomunica[TF3_C].autorizoejecucion();
                }

                if   (M[T5] && M[T1] && M[C1]){
            System.out.println (  "filósofo uno autorizado a comer") ;

            M[T5]= false;M[T1]= false;M[C1]= false;vectorcomunica[TF1_C].autorizoejecucion();
                }

                if   (M[T1] && M[T2] && M[C2]){
            System.out.println (  "filósofo dos autorizado a comer") ;

            M[T1]= false;M[T2]= false;M[C2]= false;vectorcomunica[TF2_C].autorizoejecucion();
                }



                if   (M[T3] && M[T4] && M[C4]){
            System.out.println (  "filósofo cuatro autorizado a comer") ;

            M[T3]= false;M[T4]= false;M[C4]= false;vectorcomunica[TF4_C].autorizoejecucion();
                }

                if   (M[T4] && M[T5] && M[C5]){
            System.out.println (  "filósofo cinco autorizado a comer") ;

            M[T4]= false;M[T5]= false;M[C5]= false;vectorcomunica[TF5_C].autorizoejecucion();
                }


          //try
                    //{
                    //          this.rtThread.sleep(50000);
                    //}
                    //catch(InterruptedException e){}

         }//end while true
         } //en run
public void start() {
       this.rtThread.start();
   }
}// end coordinador

class rtfilguardas{
```

```
   public static void main(String[] args) {
      //RealTimeRunner runner = new RealTimeRunner(10, 5);
      // runner.start();

   int i;

// definición de contantes para las transiciones
int TF1_P =0,TF1_C=1,TF2_P=2,TF2_C=3,TF3_P=4,TF3_C=5,TF4_P=6,TF4_C=7,TF5_P=8,TF5_C=9;

// definición de contantes para los lugares
int P1 =0,C1=1,T1=2,P2=3,C2=4,T2=5,P3=6,C3=7,T3=8,P4=9,C4=10,T4=11,P5=12,C5=13,T5=14;

// estado incial de la red de petri
boolean[] estado={true,false,true,true,false,true,true,false,true,true,false,true,true,false,true};

// vector de monitores para comunicar la acepatación del comienzo del disparo de las transiciones

System.out.println (  "hola") ;
comunica com1=new comunica();
comunica com2=new comunica();
comunica com1p=new comunica();
comunica com1c=new comunica();
comunica com2p=new comunica();
comunica com2c=new comunica();
comunica com3p=new comunica();
comunica com3c=new comunica();
comunica com4p=new comunica();
comunica com4c=new comunica();
comunica com5p=new comunica();
comunica com5c=new comunica();

comunica[] comunicaarray = {com1p,com1c,com2p,com2c,com3p,com3c,com4p,com4c,com5p,com5c};

System.out.println (  "hola1") ;

// buffer para comunicar los fin de code
BoundedBuffer buffer= new BoundedBuffer(10);

System.out.println (  "hola2") ;

transicion TF1_Ptask= new transicion(TF1_P,comunicaarray[0], buffer,15);
System.out.println (  "hola3") ;

transicion TF1_Ctask= new transicion(TF1_C,comunicaarray[1], buffer,16);
transicion TF2_Ptask= new transicion(TF2_P,comunicaarray[2], buffer,17);
transicion TF2_Ctask= new transicion(TF2_C,comunicaarray[3], buffer,18);
transicion TF3_Ptask= new transicion(TF3_P,comunicaarray[4], buffer,19);
transicion TF3_Ctask= new transicion(TF3_C,comunicaarray[5], buffer,20);
transicion TF4_Ptask= new transicion(TF4_P,comunicaarray[6], buffer,21);
transicion TF4_Ctask= new transicion(TF4_C,comunicaarray[7], buffer,22);
transicion TF5_Ptask= new transicion(TF5_P,comunicaarray[8], buffer,23);
transicion TF5_Ctask= new transicion(TF5_C,comunicaarray[9], buffer,24);
System.out.println (  "hola3") ;
```

```
transicion[] Transicionarray=
{TF1_Ptask,TF1_Ctask,TF2_Ptask,TF2_Ctask,TF3_Ptask,TF3_Ctask,TF4_Ptask,TF4_Ctask,TF5_Ptask,TF5_Ctask};
coordinador coor=new coordinador("COORDINADOR FILÓSOFOS",comunicaarray,estado,buffer);


System.out.println (  "hola4") ;


for (i = 0; i < 10; i++) {

              Transicionarray[i].start();
             }
System.out.println (  "hola5") ;
coor.start();
System.out.println (  "hola6") ;




}
}
```

```
ramonpor:~/javahecho2/ ./rtfilguardas4
hola
hola1
hola2
hola3
hola3
hola4
transición 3 empieza a ejecutarse (start)
transición 2 empieza a ejecutarse (start)
transición 1 empieza a ejecutarse (start)
transición 4 empieza a ejecutarse (start)
transición 5 empieza a ejecutarse (start)
transición 6 empieza a ejecutarse (start)
transición 7 empieza a ejecutarse (start)
transición 8 empieza a ejecutarse (start)
transición 9 empieza a ejecutarse (start)
transición 10 empieza a ejecutarse (start)
hola5
COORDINADOR FILÓSOFOS empieza ejecución
hola6
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo cinco autorizado a pensar
filósofo cinco pensando
filósofo cuatro pensando
filósofo tres pensando
filósofo dos pensando
filósofo uno pensando
final de 5 transiciones
filósofo cinco termina de pensar
filósofo cuatro termina de pensar
filósofo tres termina de pensar
filósofo dos termina de pensar
filósofo uno termina de pensar
 F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true false true true false true true false true true false true true
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo tres comiendo
filósofo uno comiendo
final de 2 transiciones
filósofo tres termina de comer
filósofo uno termina de comer
 F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
```

filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true true false true false true true true false true false true true
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true true false true false true true true false true false true true
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
filósofo cuatro termina de pensar
filósofo tres termina de comer

filósofo dos termina de pensar
filósofo uno termina de comer
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true true false true false true true true false true false true true
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true true false true false true true true false true false true true
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer

filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando
final de 4 transiciones
filósofo cuatro termina de comer
filósofo tres termina de pensar
filósofo dos termina de comer
filósofo uno termina de pensar
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
false true true true false true false true true true false true false true true
filósofo dos autorizado a pensar
filósofo cuatro autorizado a pensar
filósofo tres autorizado a comer
filósofo uno autorizado a comer
filósofo cuatro pensando
filósofo tres comiendo
filósofo dos pensando
filósofo uno comiendo
final de 4 transiciones
filósofo cuatro termina de pensar
filósofo tres termina de comer
filósofo dos termina de pensar
filósofo uno termina de comer
F1P, F1C, T1, F2P, F2C, T2, F3P, F3C, T3, F4P, F4C, T4, F5P, F5C, T5
true false true false true true true false true false true true false true true
filósofo tres autorizado a pensar
filósofo uno autorizado a pensar
filósofo dos autorizado a comer
filósofo cuatro autorizado a comer
filósofo cuatro comiendo
filósofo tres pensando
filósofo dos comiendo
filósofo uno pensando

Además se ha mejorado la impresión del marcado de la red de Petri. Ahora se imprime cuando se han ejecutado los fín de code.

Se observa que el filósofo cinco nunca come. Se queda en el estado preparado para comer. Esto es debido a la secuencialidad del coordinador en la ejecución del test de habilitación. Los tenedores están siempre activos dado que el disparo de las transiciones es prácticamente inmediato, en comparación con el ciclo del coordinador.

IMPLEMENTACIÓN PROGRAMADA DE REDES DE PETRI CON TIEMPO

6.1 CÓDIGO SECUENCIAL.

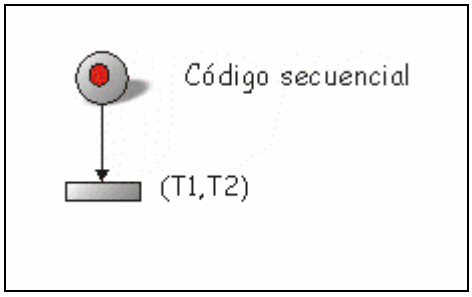

Figura 10 código secuencial.

La implementación programada del código secuencial será el propio código.

6.2 ACTIVACIÓN PERIÓDICA

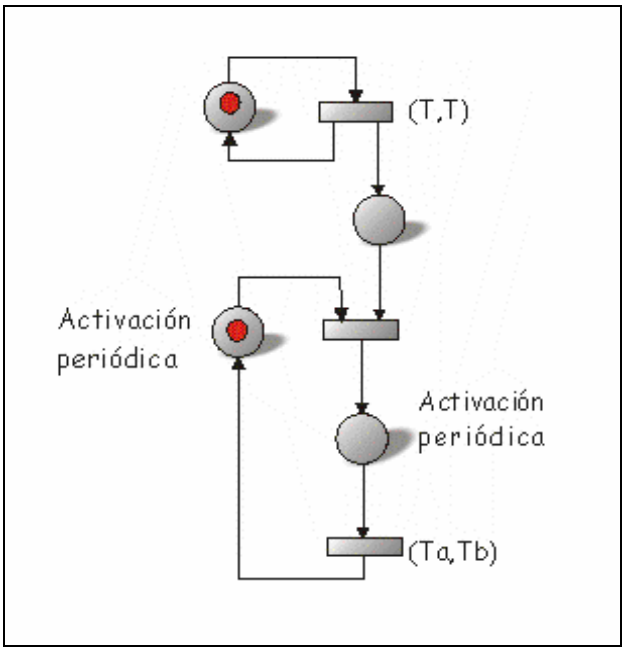

Figura 11 Activación periódica

Como en el ejemplo de los filósofos se puede configurar un thread para que se ejecute de forma periódica.

```
import javax.realtime.RealtimeThread;
import javax.realtime.PeriodicParameters;
import javax.realtime.RelativeTime;
import javax.realtime.HighResolutionClock;

public class PeriodicThreadDemo {
   public static void main(String[] args) {
      if (args.length < 3) {
         System.out.println("Usage:\n\tperiodicThredDemo <iteration> <start> <perdiod msec> \n");
         System.exit(1);
      }
      final int N = Integer.parseInt(args[0]);;
      final int start = Integer.parseInt(args[1]);
      final int period = Integer.parseInt(args[2]);

      final RelativeTime T = new RelativeTime(period, 0);

      final PeriodicParameters releaseParams = new PeriodicParameters(new RelativeTime(start, 0),T,new
RelativeTime(period/2, 0),T,null,null);

      Runnable demoLogic = new Runnable() {
            public void run() {
               RealtimeThread rtThread =RealtimeThread.currentRealtimeThread();
               boolean retVal;
               for (int i = 0; i < N; ++i) {
                  retVal = rtThread.waitForNextPeriod();
                  clock.getTime(currentTime);
                  // código periódico aquí
               }
            }
```

```
        };


      RealtimeThread rtThread = new RealtimeThread(null,releaseParams,null,null,null,demoLogic);
      rtThread.start();
      try {
         rtThread.join();
      } catch (InterruptedException e) { }
      System.out.println(">> Periodic Computation Completed");
   }
}
```

6.3 ACTIVACIÓN APERIÓDICA

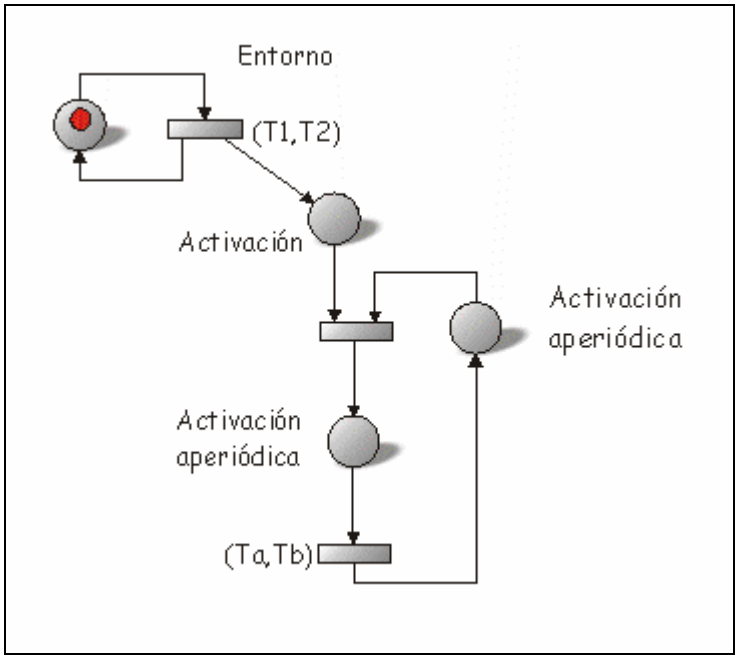

Figura 12. Activación aperiódica.

```
public class StopableThread extends RealtimeThread{
   public StopableThread (sp s, mp m){};
   public void body() throws AIE{
//código de la tarea todos los métodos deberían lanzar AIE
   }
   public void run() {try{body();}catch (AIE e){};}
}

public class ExternalEvent extends AsyncEvent{
   public void ExternalEvent() {};
   public void native BindTo() {
// código para enganchar un suceso externo a un evento (dependiente de la                    // implementación)
   }
}

public class StopIt extends AsyncEventHandler{
   RealtimeThread t; // thread que vamos a parar
   public StopIt (RealtimeThread T){t = T;}
   public void run() {t.interrupt();}
```

```
}

ExternalEvent ee = new ExternalEvent();
ee.BindTo(args); // asociar suceso externo al evento

StopIt Handler = new StopIt; // manejador de evento

StopableThread st = new StopableThread (s,m);
ee.AddHandler (Handler(st));
st.Start();
```

6.4 TIMEOUT EN EJECUCIÓN.

La ejecución de una actividad se abandona si no es finalizada antes de un tiempo determinado.

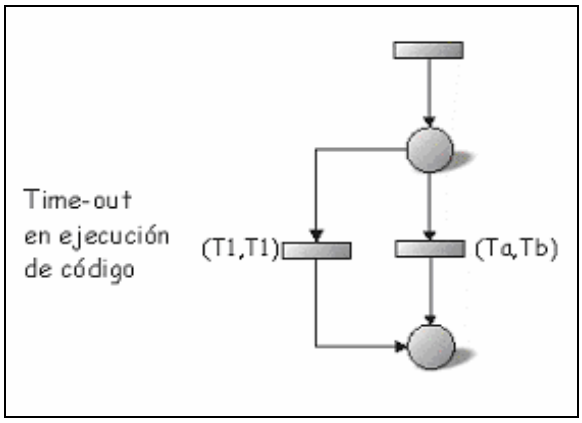

Figura 13. Time-out en ejecución de código.

Es un caso difícil de implementar en Java dado que los métodos suspend y stop han sido desechados. La única forma de realizar la implementación es mediante una transferencia asíncrona del control temporizada. El código deberá ser declarado como interrumpible

6.5 TRANSFERENCIA ASÍNCRONA DE CONTROL.

La cual permite abandonar la actividad si se produce determinado evento.

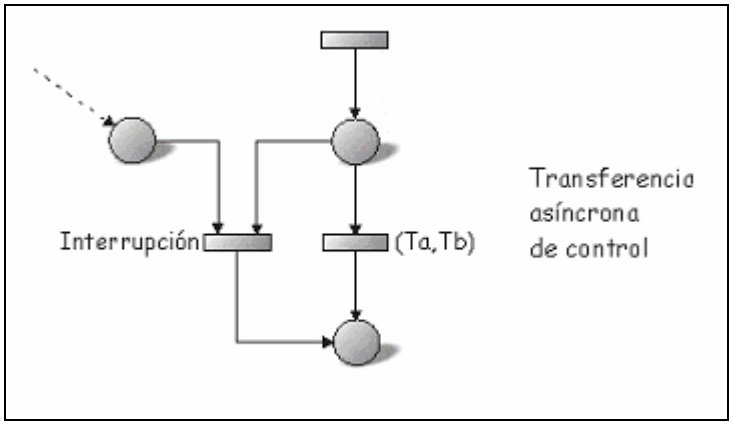

Figura 14. Transferencia asíncrona.

Por las mismas razones que el time-out, se programará como una ATC, por lo tanto el código también declararse como interrumpible

```
import javax.realtime.*;
```

```
class ATCPrueba implements Interruptible{
public void noInterrumpible(){ /* codigo no interrumpible*/ }
public void interrumpible() throws AsynchronouslyInterruptedException {
/* codigo interrumpible */ }
public void run (AsynchronouslyInterruptedException e) throws
AsynchronouslyInterruptedException {
interrumpible();
noInterrumpible ();
e.disable();
// no interrumpible
e.enable(); }
public void interruptAction (AsynchronouslyInterruptedException e){
// Acciones a ejecutar cuando se recibe la interrupción
} }

class prueba2 {
public static void main (String a[]){
ATCPrueba atc= new ATCPrueba();
Timed t= new Timed (new RelativeTime(500,0));
t.doInterruptible(atc);
}
}
```

6.6 IMPLEMETACIÓN CENTRALIZADA DE REDES DE PETRI CON TIEMPO

```
import javax.realtime.*;
import java.util.*;

// problemas
// el coordinador debe identificar el semaforo con que se comunica con  el proce
so code
// sino el coordinador llama siempre al mismo semaforo
// esto es incorrecto dado que supone un proceso al activarse en el programa el
// primero que pille el semaforo lo bloquea y pone su identificación
// al llamar a la función comienzainicio(int dato)
// al activarse los hilos proceso (star) se hace de forma secuenciada
// en el programa (atención que la ejecución es concurrente pero la activación d
e esa ejecución
// es secuenciada)

// soluciones
// creamos varios semaforos uno (al menos ) por proceso
// creamos un vector de semaforos
// quitamos del constructor del coordinador el semaforo único
class evento{
private
      transicion tr;
      AbsoluteTime hora;
      public
      evento(transicion atr,AbsoluteTime ahora){//constructor eventos
            tr=atr;
            hora=ahora;
      }
      //falta TpoEventoMasUrgente,TrEventoMasUrgente,TipoEventoUrgente
}
class transicion{
      private //ident para identificar el codigo asociado a la tr
      int prioridad,ident,Nlr;//donde esta situado su lugar representante
      boolean[] Plr;//lugares representantes marcados tras disparo
      boolean[] Ls;//lista lugares de sincronizacion
      boolean[]Pls;//lugares de sincronizacion marcados tras disparo
      RelativeTime tpo;
      char tipo;//syco "s",time "t",code "c"

      public
      //constructor
      transicion(int aprioridad,int aNlr,boolean[] aPlr,boolean[] aLs,
                  boolean[] aPls,RelativeTime atpo,char atipo,
                  int aident){
            prioridad=aprioridad;
            Nlr=aNlr;Plr=aPlr;Ls=aLs;Pls=aPls;
            tpo=atpo;
            tipo=atipo;
            ident=aident;
      }
```

```
        RelativeTime DameTpo(){
            return tpo;
    }
        char DameTipo(){//devuelve el tipo de la transicion
            return tipo;
        }
        int DameIdent(){//identifica la transicion
                return ident;
        }
        boolean Sensibilizada(){
            boolean sensi=true;
            for (int i=0;i<Ls.length;i++){
                if (Ls[i]==false)
                    sensi=false;
            }
            return sensi;
        }
        void DesmarcarLs(){//desmarco lugares de sincronizacion(entrada)
            for (int i=0;i<Ls.length;i++){
                Ls[i]=false;
            }
        }
        void MarcarPlsyPlr(){//marco lugares de sinc. y rep. marcados
            for (int i=0;i<Pls.length;i++){//tras disparo(salida)
                Pls[i]=true;
            }
            for (int i=0;i<Plr.length;i++){
                Plr[i]=true;
            }
        }
    }
    class lugar{
        private
        transicion[] tr;
        boolean marcado;
        public
        lugar(transicion[] atr,boolean amarcado){
            tr=atr;
            marcado=amarcado;
        }
        transicion[] DameTrs(){
            return tr;
        }
    }
    class comunica {
       private int valor1;
       private boolean disponible1=false;
          private int valor2;
       private boolean disponible2=false;

       public synchronized int aceptainicio() {
          while (disponible1 == false) {
             try {
                wait();
             } catch (InterruptedException e) { }
          }
```

```
        disponible1 = false;
        notifyAll();
        return valor1;
    }

    public synchronized void comienzainicio(int dato) {
        while (disponible1 == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        valor1 = dato;
        disponible1 = true;
        notifyAll();
    }

public synchronized int aceptafin() {
        while (disponible2==false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        disponible2=false;
        notifyAll();
        return valor2;
    }

    public synchronized void comienzafin(int dato) {
        while (disponible2==true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        valor2=dato;
        disponible2=true;
        notifyAll();
    }
}
class ProcesoCode extends Thread {
    private comunica c;
    private int num;

    public ProcesoCode(comunica ac, int anum) {
        c=ac;
        num=anum;
    }

    public void run() {
                while(true){
            c.comienzainicio(num);
            System.out.println("Proceso Code " +num+ " inicio ");
            System.out.println("Ejecutando codigo...."+num);
            c.comienzafin(num);
            System.out.println("Proceso Code " +num+ " fin ");
        }
    }
```

```
}

class coordinador extends Thread {
        private comunica c;
   private int num;
       private
       AbsoluteTime horaActual;
       lugar[] Lf;//listas de formacion "Lf" y de tratamiento "Lt"
       lugar[] Lt;//lista con los lugares representantes marcados
;// vector con los semaforos de los procesos
       Vector Lev=new Vector();//lista de eventos
       transicion[] tr;
       public
       //constructor coordinador
       coordinador(comunica[] cvector,lugar[] aLt){
            Lt=aLt;
       c2vector=cvector;

       }
public void run(){
            Clock clock=Clock.getRealtimeClock();
            horaActual=clock.getTime();//Cojo la hora actual
            System.out.println("La hora es "+horaActual);
       while(true){
/******************FASE DE ANALISIS DE LA SENSIBILIZACION**************/

            int o=0;//inicio contador Lf
            boolean meteLf=true;//si no trs disparables ese lugar en Lf

            for(int i=1;i<Lt.length;i++){//transiones Syco
                 tr=Lt[i].DameTrs();//trs asociadas al lugar rep.marcado
                 for(int e=0;e<tr.length;e++){
                      switch((char)tr[e].DameTipo()){
                           case 's'://transiciones syco
                           if(tr[e].Sensibilizada()){
                           meteLf=false;
                           tr[e].DesmarcarLs()    ;
                           }
                           break;
                           case 't'://transiciones time
                           if (tr[e].Sensibilizada()){//registro el evento
                           meteLf=false;
                           Lev.addElement((evento)new evento(tr[e],horaActual.add(
                                     tr[e].DameTpo())));
                           }
                           break;
                           case 'c'://transiciones code
                           int dato=0;
                           if (tr[e].Sensibilizada()){
                                //dato=c.aceptainicio();
                                dato=tr[e].DameIdent();
                                c2vector[dato].aceptainicio();
                          // if (dato==tr[e].DameIdent()){
                           System.out.println("COORDINADOR acepta inicio: "+dato);
                           tr[e].DesmarcarLs();//desmarco entradas
```

```
                        System.out.println("Insertar en lista de formacion...."+dato);
                       // }
                        }
                        break;
                        default://no corresponde con ningun tipo de tr
                        System.out.println("Error de tipo en transicion "+
                        tr[e].DameIdent());
                    }

                }
                /*if (meteLf){//si en Lr no trs disparables meto en Lf
                     Lf[o]=Lt[i];
                    o++;
                }*/
            }



/**************************FASE DE DISPARO DE TRANSICIONES**************/
      while(true){//while 2
      int dato=0;
      dato=c.aceptafin();//falta tr["7"],la x =7
   if (dato==7){//mirar si puedo implementar aqui los fin de code
      System.out.println("COORDINADOR acepta fin: " +dato);
      Clock clock1=Clock.getRealtimeClock();
           horaActual=clock1.getTime();//Cojo la hora actual
           tr[7].MarcarPlsyPlr();//marco salidas
      System.out.println("Actualizar marcado......"+dato);
   }
      }//findewhile2
      //Lt=Lf;Lf=null;//Lf en Lt para el siguiente ciclo
      }//finwhile1
      }//fin de run
}//fin coordinador

class centralizada2{
public static void main (String args[]){

//inicio de los valores
boolean[] Plra={false,false,false,false,true,false};
boolean[] Lsa={false,true,false,false,false,false};
boolean[] Plsa={false,false,false,false,false,false};
RelativeTime tpo1=new RelativeTime(0,0);
transicion ta=new transicion(10,2,Plra,Lsa,Plsa,tpo1,'s',0);
boolean[] Plrb={false,false,false,false,false,true};
boolean[] Lsb={false,false,false,true,false,false};
boolean[] Plsb={false,false,false,false,false,false};
RelativeTime tpo2=new RelativeTime(200,0);
transicion tb=new transicion(7,2,Plrb,Lsb,Plsb,tpo2,'c',1);
transicion[] tp2={ta,tb};
lugar p2=new lugar(tp2,true);
lugar[] vlug={p2};
comunica c1=new comunica();
comunica c2=new comunica();
comunica c3=new comunica();
comunica[] cvector={c1,c2,c3};
```

```
coordinador coor=new coordinador(cvector,vlug);
ProcesoCode pc1=new ProcesoCode(c1,1);
ProcesoCode pc2=new ProcesoCode(c2,2);
ProcesoCode pc3=new ProcesoCode(c3,3);

coor.start();

pc2.start();
pc3.start();
pc1.start();


}
}
```

BIBLIOGRAFÍA

Francisco José García Izquierdo. *Modelado e Implementación de Sistemas de Tiempo Real Mediante Redes de Petri con Tiempo*. Universidad de Zaragoza, Septiembre de 1999.

J.L. Villarroel. *Integración Informática del Control en Sistemas Flexibles de Fabricación*. Tesis Doctoral. Universidad de Zaragoza. 1990.

Villarroel, J., Tardós, J.D.: *Ingeniería de sistemas de tiempo real.* Apuntes del curso de doctorado. Universidad de Zaragoza.

Villarroel, J., Briz, J.L.: *Implementación programada de Redes de Petri.* Apuntes del curso de doctorado. Universidad de Zaragoza.

The Real-Time for Java expert group:*The Real-Time Specification for Java.* Copyright © 2000, Adisson-Wesley

www.rtj.org

Alan Burns and Andy Wellings. Sistemas de tiempo real y sus lenguajes de programación (tercera edición). Adisson-Wesley Iberoamericana España 2003.

http://www.cs.york.ac.uk/rts/RTSBookThirdEdition.html

Doug Lea. *Programación Concurrente en Java. Principios y patrones de diseño*. Adisson-Wesley Iberoamericana España. 2000

http://gee.cs.oswego.edu/dl/cpj/index.html

Angelo Corsaro and Douglas C. Schmidt. *The Design and Performance of the jRate Real-time Java Implementation.* The 4th International Symposium on Distributed Objects and Applications. Electrical and Computer Engineering Department. University of California, Irvine, CA 92697, October-November, 2002.

http://www.cs.wustl.edu/~corsaro/jRate/

http://www.cs.wustl.edu/~corsaro/jRate/repo/jRate/docs/api/index.html

Angelo Corsaro and Douglas C. Schmidt. *Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems.* The 8th IEEE Real-Time and Embedded Technology and Applications Symposium. Electrical and Computer Engineering Department. University of California, Irvine, CA 92697, 2002.

http://www.cs.wustl.edu/~corsaro/jRate/

Sun Microsystems. Inc. *Java Native Interface Specification*.Copyright © 1996, 1997 Sun Microsystems, Inc. 1997

http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html

http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html

Bill Venders. *Inside The Java Virtual Machine, 2nd Edition.*McGraw-Hill Professional Publishing, December (1999

http://www.artima.com/insidejvm/blurb.html

Bill Venders. *Under the Hood: How the Java virtual machine performs thread synchronization*. Copyright © 1996-2001 Artima 1997

http://www.javaworld.com/javaworld/jw-07-1997/jw-07-hood_p.html

http://www.artima.com/underthehood/thread.html

Allen Holub. *Programming Java threads in the real world, Part 1: A Java programmers guide to threading architecture*. JavaWorld, September 1998

http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads_p.html

Allen Holub. *Programming Java threads in the real world, Part 2: The perils of race conditions, deadlock, and other threading problems.* JavaWorld, October 1998

http://www.javaworld.com/javaworld/jw-10-1998/jw-10-toobox_p.html

Douglas Kramer, Bill Joy and David Spenhoff. *The Java™ Platform (a white paper).* Copyright © 1996, 1999 Sun Microsystems, Inc., May 1996

http://java.sun.com/docs/white/index.html

Sun Microsystems, Inc. *Multithreaded Programming Guide*. Copyright © 2000 Sun Microsystems, Inc.

http://docs.sun.com/?p=/doc/806-6867

Mark Secrist and Laura Smyrl. *Threads and the Java virtual machine*. AppDev ATC, December 2000. Copyright © 1994, 2002 Hewlett-Packard Company.

http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,390,00.html