Tema 12

Hilos en Java

Procesos e Hilos

En la programación concurrente, hay dos unidades básicas de ejecución: procesos e hilos. En el lenguaje de programación Java, la programación concurrente está más relacionada con los hilos. Sin embargo, los procesos también son importantes.

Un **Proceso** es un ambiente de ejecución autocontenido. Por lo general tiene recursos de ejecución completos y privados, en particular cada proceso tiene su propio espacio de memoria.

A veces se ve a los procesos como sinónimo de programas o aplicaciones. Sin embargo, una aplicación puede ser un conjunto de procesos cooperativos. Para facilitar la comunicación entre procesos, la mayoría de los sistemas operativos soportan recursos de **Comunicación entre Procesos**, IPC, como tubos y sockets. IPC no sólo se usa para comunicar procesos en un solo sistema sino procesos en diferentes sistemas.

Un **Hilo**, llamado a veces proceso ligero, provee también un ambiente de ejecución. Pero crear un nuevo hilo requiere de menos recursos que crear un nuevo proceso.

Los hilos existen dentro de los procesos – cada proceso tiene al menos un hilo. Un proceso con un solo hilo tiene las siguientes propiedades:

- El proceso empieza su ejecución en un punto bien conocido. En los programas como C, C++ o Java el proceso empieza su ejecución en la primera sentencia del método `main()`.
- La ejecución de las sentencias sigue una secuencia predefinida completamente ordenada para un conjunto de entradas.
- Durante la ejecución, el proceso tiene acceso a ciertos datos. En Java hay tres tipos de datos a los que un proceso puede acceder: variables locales que se encuentran en la pila del hilo. Variables de instancia que se accesan mediante referencias y variables estáticas que se acceden mediante clases o referencias a objetos.

En la figura 11.1 se ilustra un ejemplo de un sistema multitarea, en el cual hay varias aplicaciones en ejecución. Cada uno de ellos con un solo hilo de ejecución. En este caso tenemos varios procesos ejecutándose simultáneamente.

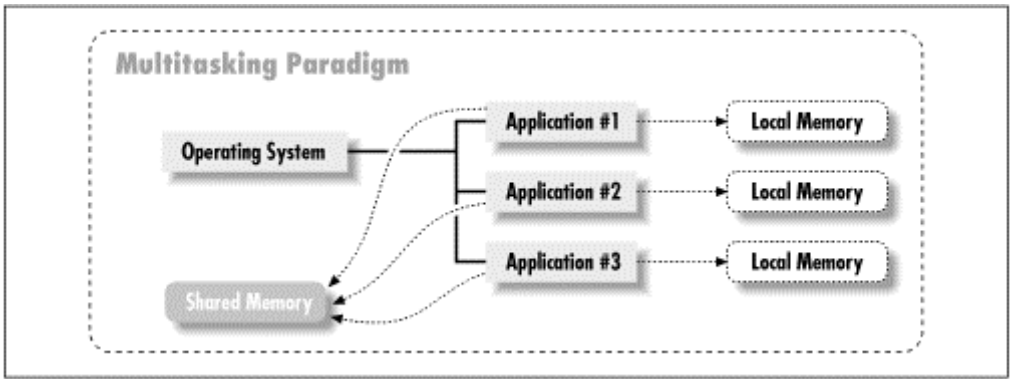

Figura 12.1

Aunque para el usuario, los procesos parecen estar ejecutándose simultáneamente, esto solo puede ocurrir en un sistema con varios procesadores o con un procesador con varios núcleos de ejecución. En el caso de un sistema con un solo procesador mononúcleo, la apariencia de ejecución simultánea se logra compartiendo el procesador entre los diferentes procesos asignándole a cada uno de ellos una ranura de tiempo y cambiando a ejecutar otro proceso cuando el tiempo de la ranura se ha agotado.

En un sistema multitarea que soporta multihilos, los hilos comparten los recursos de un proceso, incluyendo la memoria y los archivos abiertos. Esto hace la comunicación eficiente pero potencialmente problemática. En este caso cada proceso puede tener uno o más hilos. Los diferentes procesos y sus hilo pueden ejecutarse simultáneamente en forma real o aparente, dependiendo del número de procesadores o núcleos.

En un proceso, los hilos múltiples tienen las siguientes propiedades:

- Cada hilo empieza su ejecución en un punto bien conocido. Para alguno de esos hilos ese punto es la primera sentencia del método `main()`. Para el resto de los hilos, el programador decide ese punto al codificar el hilo.
- Cada hilo ejecuta su código desde su punto inicial en una forma ordenada y predefinida para un conjunto de entradas.
- Cada hilo ejecuta su código independientemente de los otros hilos en el programa. Si se requiere existen mecanismos para que los hilos cooperen entre sí.
- Los hilos aparentar ejecutar con cierto grado de simultaneidad.
- Los hilos tienen acceso a varios tipos de datos. Cada hilo tiene sus propias variables locales. Los hilos pueden compartir las variables de instancia. Los hilos comparten en forma automática a las variables estáticas.

En la figura 12.2 se ilustra un ejemplo de un sistema multitarea, en el cual hay varias aplicaciones en ejecución. Cada uno de ellos puede tener varios hilos de ejecución. En este caso, uno de esos procesos es la máquina virtual de java.

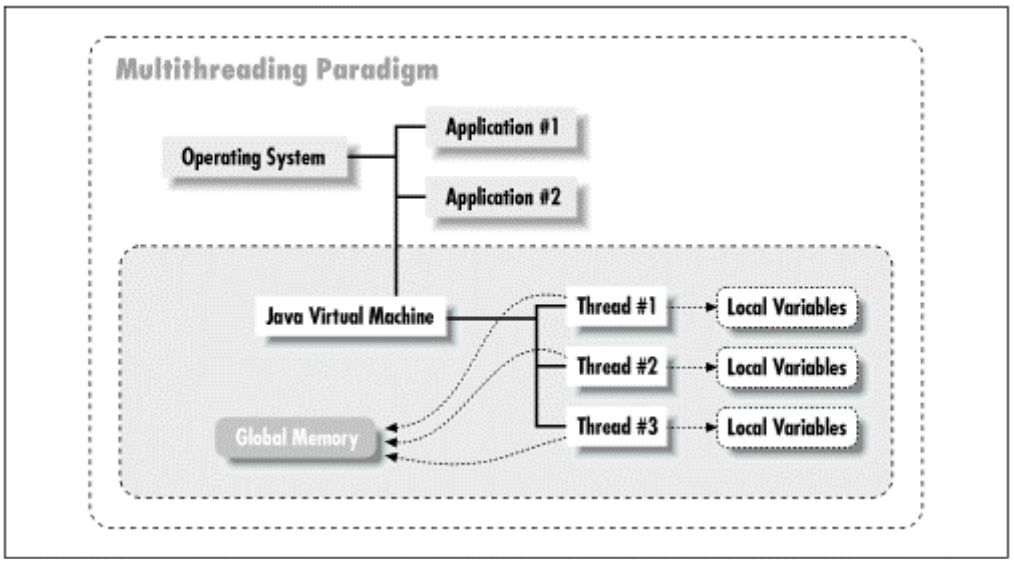

Figura 12.2

La ejecución multihilo es una característica esencial de la plataforma de Java. Cada aplicación tiene al menos un hilo o varias si se cuentan los hilos del sistema que administran la memoria y el manejo de las señales. Pero desde el punto de vista del programador se empieza con un hilo, llamado el hilo principal. Este hilo tiene la habilidad de crear hilos adicionales como se verá más adelante.

La API de Java para Hilos

Para comprender el funcionamiento de un programa multihilo veamos primero un programa con un solo hilo de ejecución. Considere el el siguiente programa formado por dos clases:

Hilo.java

```
public class Hilo {
   public void run() {
      for (int i = 0; i < 100; i++) {
         System.out.println(i + ": Hola");
      }
   }
}
```

Prueba.java

```
public class Prueba {
   public static void main(String[] args) {
      Hilo hilo = new Hilo();

      hilo.run();
   }
}
```

En la figura 12.3 se muestra el comportamiento del programa con el tiempo. En este caso como sólo hay un hilo de ejecución las sentencias se ejecutan una después de la otra. No hay concurrencia.

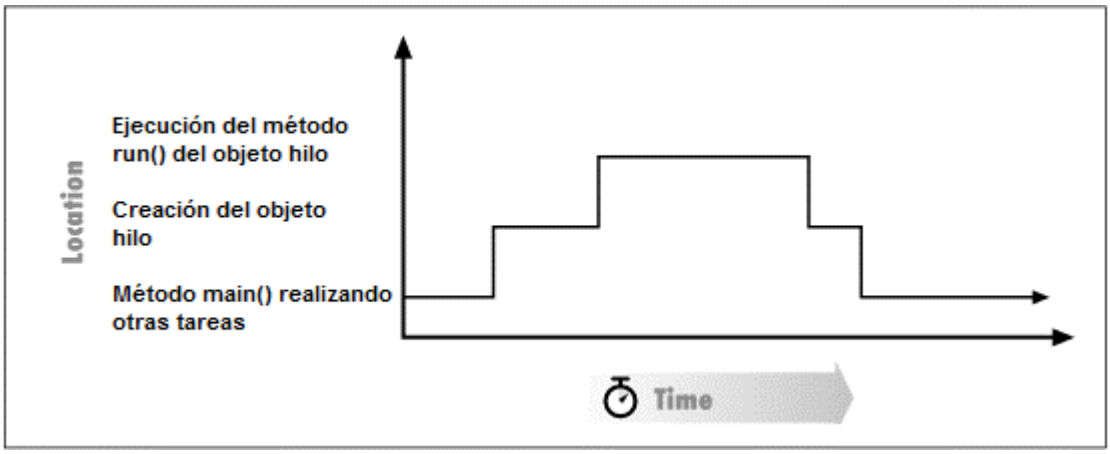

Figura 12.3

Para hacer que el método `run()` ejecute en paralelo con el método `main()` y otros métodos de la clase Prueba, hay que modificar la clase Hilo para que sea ejecutado por un nuevo hilo. Para ello nuestra clase Hilo deberá heredar de la clase `Thread`. Como se muestra en el siguiente código:

Hilo.java

```
public class Hilo extends Thread {
   public void run() {
      for (int i = 0; i < 100; i++) {
         System.out.println(I + ": Hola");
      }
   }
}
```

También debemos modificar a la clase que invoca a la clase `Hilo` para que en lugar de invocar a su método `run()` invoque a su método `start()`.

Prueba.java

```
public class Prueba {
   public static void main(String[] args) {
      Hilo hilo = new Hilo();

      hilo.start();
   }
}
```

El método `start()` es quien crea otro hilo de ejecución. Este hilo, después de un proceso de inicialización invoca al método `run()`. Este mismo hilo, cuando el método `run()` completa se encarga de ños detalles de terminación del hilo. El método `start()` del hilo original regresa inmediatamente, por lo que el método `run()` ejecutará en el nuevo hilo al mismo tiempo que el método `start()` regresa en el primer hilo como se muestra en la figura 12.4.

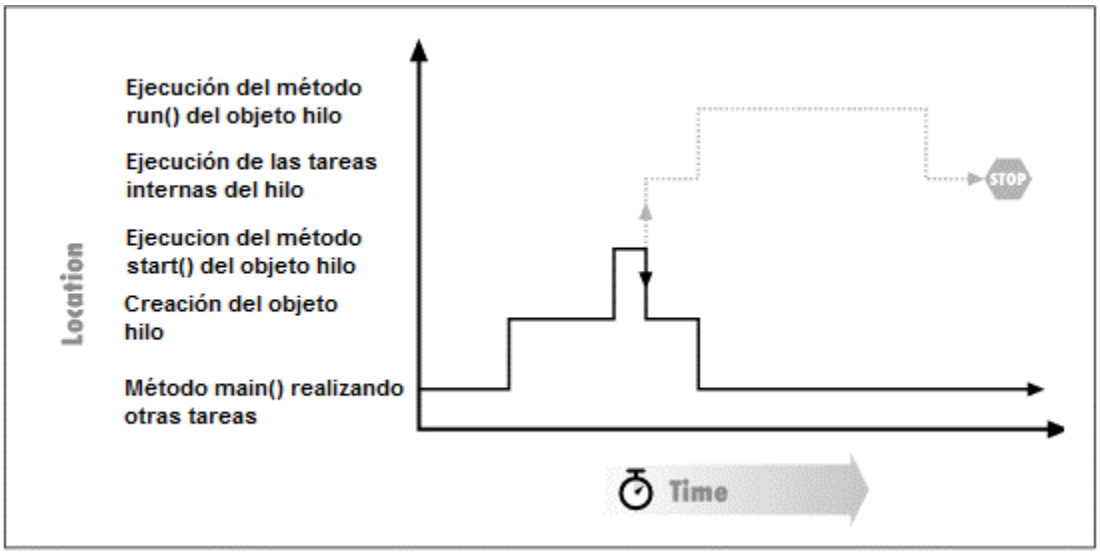

Figura 12.4

La clase `Thread` forma parte de la API de Java para hilos. Parte de esa API se muestra en la figura 12.5.

Tabla 12.1 Atributos de la Clase Thread

`public static final int` **`MAX_PRIORITY`** La máxima prioridad que puede tener un hilo
`public static final int` **`MIN_PRIORITY`** La mínima prioridad que puede tener un hilo
`public static final int NOR`**`M_PRIORITY`** La prioridad por ausencia, que puede tener un hilo

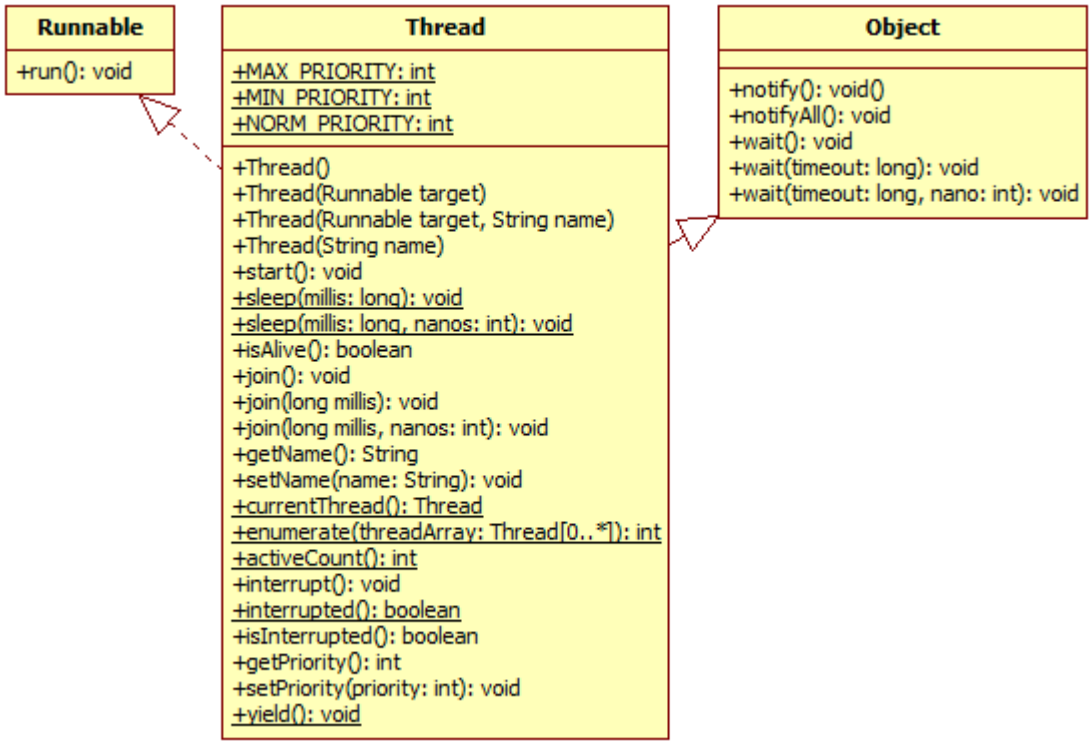

Figura 12.4

Tabla 12.2 Métodos de la Clase Thread.

`public Thread()`
`public Thread(Runnable target)`
`public Thread(Runnable target, String name)`
`public Thread(String name)`
 	
Crean nuevos hilos. El parámetro target denota el objeto del cual se ejecuta el método `run()`.El valor por omisión es `null`. El parámetro `name` establece el nombre del nuevo hilo. El valor prestablecido es "`Thread-`"*n*, donde n es un entero consecutivo.

`public void` **`start`**`()` Hace que este hilo inicie su ejecución; La máquina virtual de Java llama al método `run()` de este hilo. Como resultado hay dos hilos ejecutándose concurrentemente. El hilo actual (que regresa de la llamada al método `start()` y el otro hilo (que ejecuta su método `run()`). Es ilegal iniciar un hilo más de una vez. En particular un hilo no puede reiniciarse una vez que completa su ejecución. **Lanza**: `IllegalThreadStateException` – Si el hilo ya ha sido iniciado.
`public void` **`run`**`()` Si este hilo fue construido usando un objeto `Runnable` diferente, entonces se invoca al método `run()` del objeto. De otra forma, el método no hace nada y regresa.

Hilos Usando la Interfaz Runnable

Hay ocasiones en que crear un hilo heredando de la clase Thread no es conveniente. Por ejemplo, si la clase que deseamos que sea un hilo, es una subclase, no podemos hacer que herede de la clase Thread ya que una clase sólo puede heredar de una clase. En estos casos, en vez de hacer que la clase herede de la clase Thread haremos que la clase implemente la interfaz Runnable, como se muestra en el siguiente código:

HiloRunnable.java

```
public class HiloRunnable implements Runnable {
   public void run() {
      for (int i = 0; i < 100; i++) {
         System.out.println(i + ": Hola desde la clase Hilo");
      }
   }
}
```

Una clase que implemente la interfaz debe implementar el método run(). Sin embargo para crear un nuevo hilo no es suficiente con crear una instancia de la clase que implementa la interfaz Runnable e invocar a su método start() ya que la clase no hereda de la clase Thread y no tiene el método start(). En lugar de ello crearemos un nuevo hilo y le pasaremos al constructor de la clase Thread la instancia de la clase que implementa la interfaz Runnable, como se muestra en el siguiente código:

PruebaHiloRunnable.java

```
public class PruebaHiloRunnable {
    public static void main(String[] args) {
      Runnable hiloRunnable = new Hilo();
      Thread hilo = new Thread(hiloRunnable);

      hilo.start();
    }
}
```

La razón para pasarle un objeto Runnable al constructor de la clase Thread es para que el hilo tenga acceso al método run() del objeto Runnable para que lo ejecute. El método start() del hilo llama al método run() de la clase Thread ya que no fue sobrescrito, que a su vez llama al método run()del objeto Runnable ya que el método run() por ausencia de la clase Thread tiene la forma:

```
      public void run() {
         if (target != null) target.run();
      }
```

Donde target es el objeto `Runnable` que le pasamos al constructor del hilo. Así que el hilo empieza la ejecución de su método run() quien inmediatamente llama al método run() del objeto `Runnable`.

Suspender y Parar un Hilo

La ejecución de un hilo puede ser suspendida temporalmente, invocando a su método sleep(), tabla 12.3.

Tabla 12.3 Métodos de la Clase Thread. Cont.

`public static void` **`sleep`**`(long millis) throws InterruptedException`
`public static void` **`sleep`**`(long millis, int nanos)`
`  throws InterruptedException`
 	
Hace que el hilo, actualmente en ejecución se duerma (cese temporalmente su ejecución) por el número de milisegundos (más el número de nanosegundos, en el caso del segundo método) dado por el parámetro, sujeto a la precisión y exactitud de los temporalizadores y el programador tareas. El hilo no pierde la propiedad de ninguno de sus monitores.
 	
El valor del parámetro `nanos` debe estar en el rango de 0 – 999999 nanosegundos.
 	
Lanza:
`InterruptedException` – Si cualquier hilo ha interrumpido este hilo. El estado interrumpido del hilo actual se limpia cuando se lanza esta excepción.
`IllegalArgumentException` – Si el valor de `millis` es negativo o el valor de `nanos` no está en el rango de 0 – 999999 nanosegundos.

`public final boolean` **`isAlive`**`()` Checa si el hilo está vivo. Un hilo está vivo si ha sido iniciado y no ha muerto.

El siguiente código muestra el uso del método `sleep()` de la clase `Thread` para implementar un hilo que ejecuta una tarea periódicamente. En este ejemplo, el método suspende la ejecución del hilo por 1000 ms por lo que el ciclo despliega el mensaje aproximadamente cada 1 s mientras se repita el ciclo.

Este código también ilustra la forma de parar un hilo. Para parar un hilo hay que hacer que su método `run()` termine su ejecución y regrese. Para lograr eso en la clase principal hacemos que la variable vivo tome el valor de falso, lo que hará que el ciclo termine y termine también el método `run()`. También en la clase principal le asignamos a la referencia al hilo, `temporizador`, el valor de null para que el recolector de basura destruya al objeto y libere los recursos que el hilo tenía asignados.

Temporizador.java

```
public class Temporizador extends Thread {
    public volatile boolean vivo;

    public Temporizador() {
        vivo = true;
```

```
    }

    public void run() {
        while (vivo) {
            try {
                System.out.println("Despierto");
                sleep(1000);
            } catch (Exception e) {
            }
        }
    }
}
```

PruebaTemporizador.java

```
public class PruebaTemporizador {
    public static void main(String[] args) {
        Temporizador temporizador = new Temporizador();

        temporizador.start();

        try {
            Thread.sleep(10000);
        }
        catch(InterruptedException ie) {
            System.out.println(ie.getMessage());
        }

        temporizador.vivo = false;
        temporizador = null;
    }

}
```

El ciclo de Vida de un Hilo

En la figura 12.5 muestra el ciclo de vida de un hilo. Hay un periodo de tiempo después de que se invoca al método `start()` antes de que la máquina virtual de Java pueda arrancar al hilo. En forma similar, cuando un hilo regresa de su método run(), hay un periodo de tiempo antes que la máquina virtual de Java pueda liberar los recursos ocupados por el hilo. Este atraso ocurre por que toma tiempo arrancar o terminar un hilo; luego hay un tiempo de transición entre el estado de ejecución y el estado de parado de un hilo como se muestra en la figura 12.5.

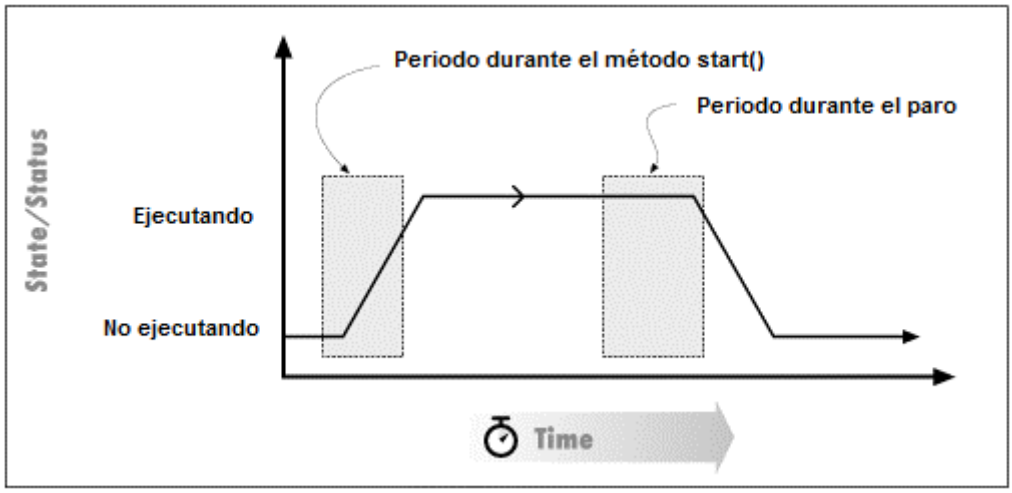

Figura 12.5

Para saber si un hilo ya ha arrancado o si ya terminado podemos llamar al método `isAlive()`. Por ejemplo, en el siguiente código, nos aseguramos que el hilo ya haya terminado antes de liberar los recursos que se le hayan asignado:

PruebaTemporizador.java

```
public class PruebaTemporizador {
    public static void main(String[] args) {
        Temporizador temporizador = new Temporizador();

        temporizador.start();

        try {
            Thread.sleep(10000);
        }
        catch(InterruptedException ie) {
            System.out.println(ie.getMessage());
        }

        temporizador.vivo = false;
        while (temporizador.isAlive()) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ie) {
                System.out.println(ie.getMessage());
            }
        }

       temporizador = null;
    }
}
```

Unión de Hilos

En el ejemplo anterior usamos los métodos `isAlive()` y `sleep()` de un hilo para esperar a que el hilo termine su ejecución antes de liberar los recusos

asignados al hilo. Hay otros métodos en la API de Java que son más adecuados para esta tarea. A este acto de esperar se le conoce como una unión de hilos. Nos estamos uniendo con el hilo creado previamente. Para unirnos con un hilo podemos usar el método `join()`, tabla 12.4.

Tabla 12.4 Métodos de la Clase Thread. Cont.

`public final void` **`join`**`()throws InterruptedException` Espera a que este hilo muera. **Lanza**: `InterruptedException` – Si cualquier hilo ha interrumpido este hilo. El estado interrumpido del hilo actual se limpia cuando se lanza esta excepción.

`public final void` **`join`**`(long millis)throws InterruptedException`
`public final void` **`join`**`(long millis, int nanos)throws InterruptedException`
 	
Espera a que este hilo muera cuando mucho el número de milisegundos (más el número de nanosegundos, en el caso del segundo método) dado por el parámetro. En el primer método, un valor del parámetro de 0 significa esperar por siempre.
 	
El valor del parámetro `nanos` debe estar en el rango de 0 – 999999 nanosegundos.
 	
Lanza:
`InterruptedException` – Si cualquier hilo ha interrumpido este hilo. El estado *interrumpido* del hilo actual se limpia cuando se lanza esta excepción.
`IllegalArgumentException` – Si el valor de `millis` es negativo o el valor de `nanos` no está en el rango de 0 – 999999 nanosegundos.

El siguiente código ilustra el uso del método `join()` para esperar que un hilo termine su ejecución:

PruebaTemporizador.java

```
public class PruebaTemporizador {
    public static void main(String[] args) {
        Temporizador temporizador = new Temporizador();

        temporizador.start();

        try {
            Thread.sleep(10000);
        }
        catch(InterruptedException ie) {
            System.out.println(ie.getMessage());
        }

        temporizador.vivo = false;

        try {
            temporizador.join();
        } catch (InterruptedException ie) {
            System.out.println(ie.getMessage());
        }
```

```
        temporizador = null;
    }
}
```

Tabla 12.5 Otros Métodos de la Clase Thread. Cont.

`public final String` **`getName`**`()` Regresa el nombre de este hilo.
`public final void` **`setName`**`(String name)` Cambia el nombre de este hilo por el valor del parámetro. **Lanza**: `SecurityException` – Si el hilo actual no puede modificar el nombre de este hilo.
`public static Thread` **`currentThread`**`()` Regresa una referencia al hilo actual en ejecución.
`public static int` **`enumerate`**`(Thread[] tarray)` Copia en el arreglo del parámetro todos los hilos activos. **Regresa:** El número de hilos colocados en el arreglo. **Lanza**: `SecurityException` – Si el hilo actual no puede modificar el nombre de este hilo.
`public static int` **`activeCount`**`()` Regresa el número de hilos actives.
`public void` **`interrupt`**`()` Interrumpe este hilo. Si este hilo está bloqueado en una invocación de los métodos `wait()` de la clase `Object` o de los métodos `join()` o `sleep()` de esta clase, entonces su estado *interrumpido* se limpia y recibirá una excepción del tipo `InterruptedException`. De otra manera el estado *interrumpido* se establecerá. **Lanza**: `SecurityException` – Si el hilo actual no puede modificar este hilo.
`public static boolean` **`interrupted`**`()` Prueba si el hilo actual ha sido interrumpido. El método limpia el estado *interrumpido* del hilo. En otras palabras, si este método se llama dos veces en consecutivas, la segunda llamada, regresará falso (a menos que el hilo actual sea interrumpido de nuevo, después de que la primera llamada haya limpiado su estado *interrumpido* y antes de que la segunda llamada lo inspeccione.
`public boolean` **`isInterrupted`**`()` Prueba si el hilo actual ha sido interrumpido. El método no modifica el estado *interrumpido* del hilo.
`public final int` **`getPriority`**`()` Regresa la prioridad de este hilo.

Tabla 12.5 Otros Métodos de la Clase Thread. Cont.

`public final void` **`setPriority`**`(int newPriority)` Cambia la prioridad de este hilo. **Lanza**: `IllegalArgumentException` – Si la prioridad no está en el rango de `MIN_PRIORITY` a `MAX_PRIORITY`. **Throws:** `IllegalArgumentException` - If the priority is not in the range `MIN_PRIORITY` to `MAX_PRIORITY`. `SecurityException` – Si el hilo actual no puede modificar este hilo.
`public static void` **`yield`**`()` Hace que el hilo en ejecución actualmente, haga una pausa temporal y permita que otro hilo ejecute.

Sincronización

Al compartir datos entre varios hilos puede ocurrir una situación llamada **condición de carrera** entre dos hilos intentando acceder al mismo dato a más o menos al mismo tiempo. Para ilustrar el problema veamos el siguiente problema:

Suponga que se desea implementar una aplicación para un cajero automático. La primera tarea es diseñar e implementar el caso de uso que permite a un usuario retirar efectivo de un cajero. El primer intento de algoritmo puede ser:

1. Verificar que el usuario tenga suficiente efectivo en su cuenta para cubrir el retiro. Si no lo tiene ir al paso 4.
2. Restar la cantidad a retirar de la cuenta del usuario
3. Entregar el dinero al usuario
4. Imprimir el recibo del usuario.

La implementación del algoritmo podría ser la siguiente:

```
public class CajeroAutomatico extends Cajero {
   public void retirar(float cantidad) {
      Cuenta c = getCuenta();

      if(c.deducir(cantidad))
         entregar(cantidad);
      imprimirRecibo();
   }

   Cuenta getCuenta() {
      ...
   }

   void entragar(float cantidad) {
      ...
```

```
   }

   void imprimirRecibo() {
      ...
   }
}

public class Cuenta {
   prívate float total;

   public boolean deducir(float t) {
      if(t <= total) {
         total -= t;
         return true;
      }

      return false;
   }
}
```

La implementación parece funcionar hasta que dos personas que tengan acceso a la misma cuenta (cuentas mancomunadas). Un día un esposo y su esposa (por separado) deciden vaciar la cuenta y por coincidencia lo intentan al mismo tiempo. En este caso tenemos una condición de carrera: si los dos usuarios retiran del banco al mismo tiempo, haciendo que los métodos se llamen al mismo tiempo, es posible que los dos cajeros confirmen que la cuenta tiene suficiente dinero y se los entregue a ambos usuarios. Lo que ocurre es que dos hilos acceden a la base de datos de la cuenta al mismo tiempo.

La condición de carrera ocurre porque la acción de verificar la cuenta y cambiar su estado no es una operación atómica. El hilo del esposo y el hilo de la esposa compiten por la cuenta:

1. El hilo del esposo empieza a ejecutar el método `deducir()`.
2. El hilo del esposo confirma que la cantidad a deducir es menor o igual al total de la cuenta.
3. El hilo de la esposa empieza a ejecutar el método `deducir()`.
4. El hilo de la esposa confirma que la cantidad a deducir es menor o igual al total de la cuenta.
5. El hilo de la esposa ejecuta la sentencia de substraer para deducir la cantidad, regresa verdadero y el cajero entrega el dinero.
6. El hilo del esposo ejecuta la sentencia de substraer para deducir la cantidad, regresa verdadero y el cajero entrega el dinero.

Para tratar con este problema, el lenguaje Java provee de la palabra reservada `syncronized` que impide que dos hilos ejecuten el mismo código al mismo tiempo. Si modificamos el método `deducir()` de la clase cuenta de la siguiente forma:

```
public class Cuenta {
   prívate float total;

   public syncronized boolean deducir(float t) {
      if(t <= total) {
         total -= t;
         return true;
      }

      return false;
   }
}
```

Método Sincronizado

En el ejemplo anterior el método `deducir()` está sincronizado. Para que un método sincronizado pueda ejecutarse, primero debe adquirir un token o candado. Una vez que el método adquiere ese candado, ejecuta su código y al terminar regresa el candado, sin importar la forma en que termine su ejecución, incluyendo vía una excepción. Solo hay un candado por objeto, así que si dos hilos separados tratan de llamar a un método sincronizado de un mismo objeto, sólo uno de ellos puede ejecutar el método inmediatamente; el otro hilo tiene que esperar a que el primer hilo libere el candado antes de que pueda ejecutar el método. La sintaxis de un método sincronizado es la siguiente:

```
modificadorAcceso syncronized tipo nomMetodo(lista de parámetros) {
   declaraciones

   sentencias
}
```

Bloque Sincronizado

En lugar de sincronizar todo un método podemos sincronizar un bloque. Al igual que con un método sincronizado, un bloque sincronizado debe adquirir primero un candado para poder ejecutarse, al terminar la ejecución del bloque, regresa el candado. Por lo tanto si hay dos hilos tratando de ejecutar el bloque, sólo uno de los hilos lo ejecutará inmediatamente, el otro hilo deberá esperar a que el primer hilo termine la ejecución del bloque y libere el candado para que pueda ejecutar el bloque. La sintaxis de un bloque sincronizado es la siguiente:

```
syncronized(expresión) {
   declaraciones

   sentencias
}
```

Donde expresión debe ser una expresión que se evalué a un objeto, el objeto del que se obtiene el candado.

Declarar un método como sincronizado es equivalente a:

```
modificadorAcceso tipo nomMetodo(lista de parámetros) {
   syncronized(this) {
      declaraciones

      sentencias
   }
}
```

Esperar y Notificar

La clase `Object` tiene los métodos `wait()` y `notify()`. Esos métodos permiten que un hilo libere el candado en un momento arbitrario, y espere a que otro hilo se lo regrese antes de continuar. Estas actividades deben ocurrir dentro de bloques sincronizados.

Tabla 12.4 Métodos de la Clase `Object` Empleados con Hilos

`public final void` **`notify`**`()`

Despierta un solo hilo que está esperando por el monitor de este objeto. Si hay varios hilos esperando por este objeto, uno de ellos es seleccionado para despertarse. La selección es arbitraria y ocurre a la discreción de la implementación. Un hilo espera por el monitor del objeto al llamar a alguno de sus métodos `wait()`.

El hilo despertado no podrá proceder hasta que el hilo actual libere el candado sobre el objeto. El hilo despertado competirá con cualquier otro hilo que este compitiendo activamente para sincronizarse con este objeto; por ejemplo, el hilo despertado no tiene privilegios o desventajas para ser el siguiente hilo en ponerle un candado al objeto.

Este método sólo debe llamarlo un hilo que sea el dueño del monitor del objeto. Un hilo se vuelve el dueño del monitor del objeto en una de tres formas:

- Ejecutando una método de instancia sincronizado del objeto.
- Ejecutando el cuerpo de una sentencia sincronizada que sincroniza en el objeto.
- En los objetos de tipo Class, ejecutando un método estático de la clase.

Sólo uno de los hilos a la vez puede ser el dueño del monitor del objeto.

Lanza:

`IllegalMonitorStateException` – Si el hilo actual no es el dueño del monitor del objeto.

Tabla 12.4 Métodos de la Clase `Object` Empleados con Hilos. Cont.

`public final void` **`notifyAll`**`()` Despierta a todos los hilos que están esperando por el monitor del objeto. Un hilo espera por el monitor del objeto al llamar a alguno de sus métodos `wait()`. Los hilos despertados no podrán proceder hasta que el hilo actual libere el candado sobre el objeto. Los hilos despertados competirán con cualquier otro hilo que este compitiendo activamente para sincronizarse con este objeto; por ejemplo, el hilo despertado no tiene privilegios o desventajas para ser el siguiente hilo en ponerle un candado al objeto. Este método sólo debe llamarlo un hilo que sea el dueño del monitor del objeto. **Lanza**: `IllegalMonitorStateException` – Si el hilo actual no es el dueño del monitor del objeto.
`public final void` **`wait`**`(long timeout) throws InterruptedException` Hace que el hilo actual espere hasta que otro hilo invoque al método `notify()` o al método `notifyAll()` de este objeto, o la cantidad de milisegundos especificada por el parámetro haya transcurrido. El hilo actual debe poseer el monitor de este objeto. Este método hace que el hilo actual se coloque en el estado de espera para este objeto y luego libere cualquier derecho de sincronización en este objeto. El hilo deja de estar disponible para el planificador de procesos y permanece dormido hasta que alguna de estas cuatro cosas suceden: • Algún otro hilo invoca al método `notify()` para este objeto y el hilo es seleccionado arbitrariamente para ser el hilo a despertar. • Algún otro hilo invoca al método `notifyAll()` para este objeto. • Algún otro hilo interrumpe al hilo. • El tiempo en milisegundos especificado por el parámetro ha transcurrido (más o menos). Si `timeout` es cero, el hilo espera hasta ser notificado. El hilo es removido del estado de espera para este objeto y queda disponible para el planificador de procesos. Luego compite con los otros hilos por el derecho de sincronizarse con el objeto. Una vez que ha ganado el control del objeto, todos sus derechos de sincronización en el objeto se restablecen al estado que tenían en el momento en que se invoco al método `wait()`. El hilo luego regresa del método `wait()`. **Lanza**: `IllegalArgumentException` – Si el valor de `timeout` es negativo. `IllegalMonitorStateException` – Si el hilo actual no es el dueño del monitor del objeto. `InterruptedException` – Si cualquier hilo ha interrumpido el corriente hilo antes o mientras el hilo actual está esperando por una notificación. El estado de interrumpido del hilo actual se limpia cuando se lanza la excepción.

Tabla 12.4 Métodos de la Clase `Object` Empleados con Hilos. Cont.

`public final void wait(long timeout, int nanos)`
`  throws InterruptedException`
 	
Hace que el hilo actual espere hasta que otro hilo invoque al método `notify()` o al método `notifyAll()` de este objeto, u otro hilo interrumpa al hilo actual o la cantidad de tiempo dado por sus parámetros haya transcurrido.
 	
El método es similar al método wait con un solo parámetro, pero permite un control más fino sobre la cantidad de tiempo a esperar por la notificación antes de continuar. El tiempo a esperar medido en nanosegundos está dada por:
 	
`1000000*timeout+nanos`
 	
En todos los demás aspectos, este método es similar al método `wait (long timeout)`. En particular `wait(0, 0)` significa lo mismo que `wait(0)`.

`public final void wait() throws InterruptedException` Este método se comporta como si se llamara a `wait(0)`.

Al ejecutar el método `wait()` desde un bloque sincronizado, un hilo libera el candado y se duerme. Un hilo puede hacer eso si necesita esperar que pase algo en otra parte de la aplicación. Más tarde cuando el evento esperado pase, el hilo que está en ejecución llama al método `notify()` desde un bloque sincronizado en el mismo objeto. Entonces el primer hilo se despierta y empieza a tratar de adquirir el candado otra vez.

Cuando el primer hilo logra obtener de nuevo el candado, continúa desde el punto en que se quedó. Sin embargo, el hilo que estaba esperando puede no adquirir el candado inmediatamente (o tal vez, nunca). Depende de cuando el segundo hilo eventualmente libere el candado, y que hilo logre atraparlo enseguida. El primer hilo no se despertará a menos que otro hilo llame al método `notify()` . Hay una versión sobrecargada de `wait()`, que permite especificar un tiempo de expiración. Si otro hilo no invoca al método `notify()` en el tiempo especificado, el hilo se despierta automáticamente.

Para cada llamada a `notify()`, Java despierta uno de los métodos dormidos en la llamada a `wait()`. Si hay varios métodos esperando, Java selecciona al primer hilo en base a primero en entrar – primero en salir.

La clase `Object` también tiene el método `notifyAll()` que permite despertar a todos los hilos esperando. En la mayoría de los casos se desea llamar a `notifyAll()` en vez de `notify()`.

Ejemplo sobre Esperar y Notificar

Considere el siguiente ejemplo. Se quiere simular el comportamiento de una cola a la que llegan clientes a ser atendidos. Los clientes son atendidos en una o varias estaciones. Para modelar la atención que recibe el cliente se tiene la clase `Tarea`:

Tarea.java

```
/*
 * Tarea.java
 *
 * @author mdomitsu
 */
package cola;

import java.util.Date;

public class Tarea {
    int numTarea;
    Date timeStamp;
    int duracion;

    public Tarea(int numTarea) {
        this.numTarea = numTarea;
        timeStamp = new Date();
        duracion = GeneradorAleatorio.getPoisson(1);
    }

    public int getDuracion() {
        return duracion;
    }

    public int getNumTarea() {
        return numTarea;
    }

    public Date getTimeStamp() {
        return timeStamp;
    }

    @Override
    public String toString() {
        return "Tarea " + numTarea + ": " + duracion + ", "
                + timeStamp.toString();
    }
}
```

El atributo `duracion` de la clase `Tarea`, modela el tiempo que el cliente tarda en ser atendido. Para establecer ese tiempo se utiliza un generador de números aleatorios que tiene una distribución de probabilidad de Poisson. Su código se encuentra en la clase `GeneradorAleatorio`.

GeneradorAleatorio.java

```
/*
 * GeneradorAleatorio.java
 *
 * @author mdomitsu
 */
package cola;

public class GeneradorAleatorio {
    public static int getPoisson(double lambda) {
        double L = Math.exp(-lambda);
        double p = 1.0;
        int k = 0;

        do {
            k++;
            p *= Math.random();
        } while (p > L);

        return k - 1;
    }
}
```

La clase `ColaTareas` representa la cola a la que llegan los clientes en espera de ser atendidos. La cola tiene un tamaño máximo de 5. Si la cola está llena, los clientes deben esperan a que haya espacio para entrar a la cola.

ColaTareas.java

```
/*
 * ColaTareas.java
 *
 * @author mdomitsu
 */
package cola;

import java.util.ArrayList;
import java.util.List;

public class ColaTareas {
    private List<Tarea> colaTareas = new ArrayList<Tarea>();
    private static final int MAXQUEUE = 5;

    public synchronized void ponTarea(Tarea tarea)
            throws InterruptedException {

        while (colaTareas.size() == MAXQUEUE) {
            wait();
        }

        colaTareas.add(tarea);
        notify();
    }

    public synchronized Tarea obtenTarea() throws InterruptedException {
        notify();
```

```
        while (colaTareas.size() == 0) {
            wait();
        }
        Tarea tarea = colaTareas.get(0);
        colaTareas.remove(tarea);

        return tarea;
    }

    public synchronized int getTamano() {
        return colaTareas.size();
    }
}
```

Para modelar el arribo de los clientes se tiene el método `produceTarea()` que agrega un cliente a la cola. Para establecer ese tiempo entre arribos de los clientes se utiliza un generador de números aleatorios que tiene una distribución de probabilidad de Poisson. Los clientes llegan más rápido de lo que pueden ser atendidos.

Productor.java

```
/*
 * Productor.java
 *
 * @author mdomitsu
 */
package hilos;

import cola.ColaTareas;
import cola.GeneradorAleatorio;
import cola.Tarea;

public class Productor extends Thread {
    private static final int MAX_TAREAS = 10;
    private ColaTareas colaTareas;

    public Productor(ColaTareas colaTareas) {
        this.colaTareas = colaTareas;
    }

    @Override
    public void run() {
        int i = 0;
        try {
            while (i < MAX_TAREAS) {
                produceTarea(i);

                // Simula el tiempo entre llegadas
                sleep(1000 * GeneradorAleatorio.getPoisson(0.5));
                i++;
            }
        } catch (InterruptedException e) {
        }
    }
```

```
    private void produceTarea(int i) throws InterruptedException {
        Tarea tarea = new Tarea(i);

        colaTareas.ponTarea(tarea);
        System.out.println("Productor: " + tarea + ", "
           + colaTareas.getTamano());
    }
}
```

Para modelar la atención de los clientes se tiene la clase `Consumidor`.

Consumidor.java

```
/*
 * Consumidor.java
 *
 * @author mdomitsu
 */
package hilos;

import cola.ColaTareas;
import cola.Tarea;

public class Consumidor extends Thread {
    private ColaTareas colaTareas;

    public Consumidor(String nombre, ColaTareas colaTareas) {
        super(nombre);
        this.colaTareas = colaTareas;
    }

    @Override
    public void run() {
        try {
            while (true) {
                Tarea tarea = consumeTarea();
                sleep(1000 * tarea.getDuracion());
            }
        } catch (InterruptedException e) {}
    }

    public Tarea consumeTarea() throws InterruptedException {
        Tarea tarea;

        tarea = colaTareas.obtenTarea();
        System.out.println(getName() + ": " + tarea+ ", "
                + colaTareas.getTamano());

        return tarea;
    }
}
```

En la siguiente clase de prueba se simula el caso de que hay una solo estación de atención para los clientes:

Prueba.java

```
/*
 * Prueba.java
 *
 * @author mdomitsu
 */

package pruebas;

import cola.ColaTareas;
import hilos.Consumidor;
import hilos.Productor;

/**
 *
 * @author mdomitsu
 */
public class Prueba {
    public static void main(String[] args) {
        ColaTareas colaMensajes = new ColaTareas();

        Productor productor = new Productor(colaMensajes);
        Consumidor consumidor1 = new Consumidor("Consumidor 1",
                                                colaMensajes);

        productor.start();
        consumidor1.start();
    }
}
```

Lo siguiente es una corrida del programa anterior.

```
Productor: Tarea 0: 1, Fri Apr 29 00:24:37 MST 2011, 0
Productor: Tarea 1: 3, Fri Apr 29 00:24:37 MST 2011, 1
Productor: Tarea 2: 0, Fri Apr 29 00:24:37 MST 2011, 2
Productor: Tarea 3: 0, Fri Apr 29 00:24:37 MST 2011, 3
Consumidor 1: Tarea 0: 1, Fri Apr 29 00:24:37 MST 2011, 3
Consumidor 1: Tarea 1: 3, Fri Apr 29 00:24:37 MST 2011, 2
Productor: Tarea 4: 1, Fri Apr 29 00:24:39 MST 2011, 3
Productor: Tarea 5: 1, Fri Apr 29 00:24:40 MST 2011, 4
Productor: Tarea 6: 0, Fri Apr 29 00:24:40 MST 2011, 5
Consumidor 1: Tarea 2: 0, Fri Apr 29 00:24:37 MST 2011, 4
Consumidor 1: Tarea 3: 0, Fri Apr 29 00:24:37 MST 2011, 3
Consumidor 1: Tarea 4: 1, Fri Apr 29 00:24:39 MST 2011, 2
Productor: Tarea 7: 1, Fri Apr 29 00:24:40 MST 2011, 3
Productor: Tarea 8: 0, Fri Apr 29 00:24:42 MST 2011, 3
Consumidor 1: Tarea 5: 1, Fri Apr 29 00:24:40 MST 2011, 3
Consumidor 1: Tarea 6: 0, Fri Apr 29 00:24:40 MST 2011, 3
Productor: Tarea 9: 1, Fri Apr 29 00:24:43 MST 2011, 3
Consumidor 1: Tarea 7: 1, Fri Apr 29 00:24:40 MST 2011, 2
Consumidor 1: Tarea 8: 0, Fri Apr 29 00:24:42 MST 2011, 1
Consumidor 1: Tarea 9: 1, Fri Apr 29 00:24:43 MST 2011, 0
```

En la siguiente clase de prueba se simula el caso de que hay dos estaciones de atención para los clientes:

Prueba.java

```
/*
 * Prueba.java
 *
 * @author mdomitsu
 */

package pruebas;

import cola.ColaTareas;
import hilos.Consumidor;
import hilos.Productor;

/**
 *
 * @author mdomitsu
 */
public class Prueba {
    public static void main(String[] args) {
        ColaTareas colaMensajes = new ColaTareas();

        Productor productor = new Productor(colaMensajes);
        Consumidor consumidor1 = new Consumidor("Consumidor 1",
                                                colaMensajes);
        Consumidor consumidor2 = new Consumidor("Consumidor 2",
                                                colaMensajes);


        productor.start();
        consumidor1.start();
        consumidor2.start();
    }
}
```

Lo siguiente es una corrida del programa anterior.

```
Productor: Tarea 0: 1, Fri Apr 29 00:26:58 MST 2011, 0
Consumidor 2: Tarea 0: 1, Fri Apr 29 00:26:58 MST 2011, 0
Consumidor 1: Tarea 1: 4, Fri Apr 29 00:26:59 MST 2011, 0
Productor: Tarea 1: 4, Fri Apr 29 00:26:59 MST 2011, 0
Productor: Tarea 2: 2, Fri Apr 29 00:27:00 MST 2011, 0
Consumidor 2: Tarea 2: 2, Fri Apr 29 00:27:00 MST 2011, 0
Productor: Tarea 3: 0, Fri Apr 29 00:27:01 MST 2011, 1
Productor: Tarea 4: 0, Fri Apr 29 00:27:01 MST 2011, 2
Productor: Tarea 5: 0, Fri Apr 29 00:27:01 MST 2011, 3
Productor: Tarea 6: 2, Fri Apr 29 00:27:01 MST 2011, 4
Productor: Tarea 7: 0, Fri Apr 29 00:27:01 MST 2011, 5
Consumidor 2: Tarea 3: 0, Fri Apr 29 00:27:01 MST 2011, 4
Consumidor 2: Tarea 4: 0, Fri Apr 29 00:27:01 MST 2011, 3
Consumidor 2: Tarea 5: 0, Fri Apr 29 00:27:01 MST 2011, 2
Consumidor 2: Tarea 6: 2, Fri Apr 29 00:27:01 MST 2011, 1
Consumidor 1: Tarea 7: 0, Fri Apr 29 00:27:01 MST 2011, 0
Productor: Tarea 8: 2, Fri Apr 29 00:27:03 MST 2011, 0
Productor: Tarea 9: 0, Fri Apr 29 00:27:03 MST 2011, 1
Consumidor 1: Tarea 8: 2, Fri Apr 29 00:27:03 MST 2011, 0
Consumidor 2: Tarea 9: 0, Fri Apr 29 00:27:03 MST 2011, 0
```