◆ Sistemas de Tiempo Real

Real-Time Java

José F. Ruiz
DIT/UPM

Índice

- Introducción
- Objetivos
- Áreas mejoradas en la especificación de tiempo real
- Manejo del tiempo
- Conclusiones
- Referencias

©2001-2002 José F. Ruiz

¿Por qué Java?

dit
UPM

- Simplicidad
  - ✓ Java es fácil de aprender y de usar
- Orientación a objetos
  - ✓ metodología que ha demostrado que aumenta la productividad
- Seguridad
  - ✓ no existen punteros
- Portabilidad
  - ✓ la máquina virtual de Java está disponible para una gran cantidad de plataformas
- Bibliotecas
  - ✓ funcionalidad de todo tipo
- Lenguaje muy extendido
  - ✓ facilita el soporte y mantenimiento de productos con largo ciclo de vida

Desarrollo de productos más rápido y de menor coste

©2001-2002 José F. Ruiz

Problemas de Java

dit
UPM

- Principales problemas de Java para escribir código de tiempo real:
  - ✓ dificulta la predecibilidad
    - » recolector de basura (*Garbage Collector*)
  - ✓ reduce la eficiencia
    - » lenguaje interpretado
  - ✓ su especificación no es suficientemente precisa

©2001-2002 José F. Ruiz

Dos propuestas diferentes

dit
UPM

- *The Real-Time for Java Expert Group*
  - ✓ liderado por *Sun Microsystems*
  - ✓ intenta dar soporte genérico a un amplio rango de sistemas empotrados de tiempo real
- *J Consortium*
  - ✓ proponen una extensión al núcleo de Java
  - ✓ más enfocado hacia sistemas de alta integridad mediante el *High Integrity Profile*

©2001-2002 José F. Ruiz

Requisitos iniciales

dit
UPM

- Compatibilidad hacia atrás
- Adecuarse a cualquier plataforma Java
- WOCRAC (≈ WORA)
  - ✓ *Write Once* ***Carefully*** *Run Anywhere* ***Conditionally***
  - ✓ es más importante la predecibilidad que la portabilidad
- Soportar la práctica actual y permitir futuros avances
- Predecibilidad
- Evitar extensiones sintácticas al lenguaje Java
- Permitir cierta flexibilidad en la implementación
  - ✓ equipos de telecomunicación, robots, controladores industriales, ...

©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- **Planificación**
- Manejo de memoria
- Sincronización
- Eventos asíncronos
- Transferencia asíncrona de control (ATC)
- Terminación de tareas
- Acceso a memoria física

©2001-2002 José F. Ruiz

Planificación

dit *UPM*

- Hay una gran variedad de algoritmos de planificación
  - ✓ adecuados para distintos tipos de sistemas
    - » tiempo real crítico, multimedia, ...
- Objetivo
  - ✓ escribir una especificación que se adapte a los algoritmos de planificación proporcionado por los SO existentes
    - » por ejemplo, RMS
  - ✓ especificación capaz de adaptarse a futuros algoritmos
    - » por ejemplo, EDF
- Mínimo obligado por la especificación
  - ✓ prioridades fijas, con desalojo y al menos 28 niveles de prioridad

Planificación (II)

dit UPM

- Introducción del concepto de objeto planificable
  - ✓ ser planificable es un atributo del objeto
  - ✓ interfaz *Schedulable*
    - » Cualquier objeto que implemente esta interfaz es planificado por el planificador

      threads de tiempo real y manejadores de eventos asíncronos
- Tres clases de objetos planificables (extensión de *java.lang.Thread*)

  RealtimeThread
  ↳ *NoHeapRealtimeThread* } ***threads* de tiempo real**

  AsyncEventHandler } **manejadores de eventos asíncronos**
- Mantiene compatibilidad hacia atrás con la planificación existente en Java

©2001-2002 José F. Ruiz

Planificador

dit
UPM

- El planificador es un objeto de la clase *Scheduler*
- Una instancia de esta clase por cada JVM
- Contiene los métodos para admisión de *threads*, mecanismos de manejo de eventos asíncronos, ...
- Subclases derivadas de *Scheduler* implementan algoritmos alterativos de planificación

©2001-2002 José F. Ruiz

Planificador (II)

dit UPM

- Por defecto *PriorityScheduler*
  - ✓ requerido por la especificación
  - ✓ planificación por prioridades fijas
    - » la prioridad de un objeto planificable no cambia, excepto por el protocolo de control de inversión de prioridades
  - ✓ 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
  - ✓ al menos 28 niveles de prioridad para *threads* de tiempo real
- La especificación permite incluir nuevas políticas de planificación, por ejemplo *EDFScheduler*

©2001-2002 José F. Ruiz

Parámetros de los *threads*

dit
UPM

- Parámetros usados para especificar los requisitos temporales y demanda de recursos de un *RealTimeThread (o NoHeapRealtimeThread)*
  - ✓ *ReleaseParameters*
    - » establecer plazos, *budgets*, periodo, tiempo de ejecución, ...
  - ✓ *SchedulingParameters*
    - » prioridad, importancia, ...
  - ✓ *MemoryParameters*
    - » límites en el uso de ciertas áreas de memoria
- También permite a los desarrolladores especificar la respuesta del thread ante situaciones imprevistas.
  - ✓ por ejemplo, incumplimiento de plazo de respuesta

©2001-2002 José F. Ruiz

Tareas periódicas

dit UPM

- RT-Java proporciona al programador un nivel de abstracción bastante alto. Parámetros que se pueden especificar son:
  - ➔ *cost*. El sistema garantiza que no se va a dejar a esta tarea consumir más CPU de los indicado *(budget)*
  - ➔ *importance*. Criterio de selección entre tareas de igual prioridad
  - ➔ *deadline*. Plazo de respuesta
  - ➔ *start.* Comienzo del primer periodo
  - ➔ *reStart*. Comienzo tras un *overrun*
  - ➔ *period*. Periodo de ejecución del método *run*
  - ➔ *overrunHandler*. Se ejecuta si el método *run* está aun activo al consumirse todo el *cost*.
  - ➔ *deadlineHandler*. Se ejecuta si el método *run* está aun activo cuando se cumple el plazo de respuesta

Tareas periódicas (ejemplo)

dit UPM

```
public class Periodic extends NoHeapRealtimeThread {
   public Periodic (SchedulingParameters sp, MemoryParameters mp) {}
   public void run() {
      //  Código a ejecutar por la tarea periódica
   }

// Periodo 50 ms y un budget de 2 ms por activación
PeriodicParameters p = new PeriodicParameters();
p.cost = new RelativeTime(2,0);
p.period = new RelativeTime(50,0);

// Usa una ScopedMemory de 16kB de tiempo de creación de objetos lineal
LTMemory ltm = new LTMemory (16 * 1024);
MemoryParameters m = new MemoryParameters(ltm);

Periodic t1 = new Periodic(p,m);

t1.start();
```


©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- Planificación
- **Manejo de memoria**
- Sincronización
- Eventos asíncronos
- Transferencia asíncrona de control (ATC)
- Terminación de tareas
- Acceso a memoria física

©2001-2002 José F. Ruiz

Manejo de memoria

dit UPM

- Gestión automática de memoria en Java es muy útil
  - ✓ sin embargo, resta predecibilidad al sistema
  - ✓ aunque algunos algoritmos de recolección de basura (GC) son aplicables a determinados STR
- RT-Java define extensiones al modelo de memoria
  - ✓ permite uso de memoria determinista
- Requisitos de la especificación de tiempo real
  - ✓ ser independiente de algoritmos particulares de GC
  - ✓ permitir una caracterización precisa del efecto del algoritmo concreto implementado en el sistema sobre los *threads* (tiempo de ejecución, bloqueos, ...)
  - ✓ permitir creación y destrucción de objetos sin interferencia por parte del algoritmo de GC

©2001-2002 José F. Ruiz

Manejo de memoria: requisitos

dit
UPM

- Compatibilidad hacia atrás
  - ✓ El *heap* y el GC de Java siguen existiendo
- WOCRAC
  - ✓ cualquier implementación debe soportar obligatoriamente las nuevas áreas de memoria
- Soporta la práctica actual
  - ✓ por ejemplo, preasignación de objetos y *buffers*
- Implementaciones avanzadas
  - ✓ permite la implementación de nuevos mecanismos de GC

©2001-2002 José F. Ruiz

Manejo de memoria: ideas básicas

dit UPM

- RTSJ cambia la idea de tiempo de vida de un objeto (cuando un objeto es candidato para ser borrado)
  - ✓ Manual. Tiempo de vida controlado por el programa
  - ✓ Automático. Tiempo de vida controlado por visibilidad
- Concepto de áreas de memoria (*MemoryArea*).
- 4 tipos básicos:
  - ✓ *HeapMemory.*
  - ✓ *ImmortalMemory*
  - ✓ *ScopedMemory*
  - ✓ *ImmortalPhysicalMemory*
- Caracterización del comportamiento del GC

©2001-2002 José F. Ruiz

HeapMemory

dit UPM

- Objetos en esta área se crean en el *heap*
  - ✓ equivalente al *heap* de Java
- Una instancia por cada JVM
- Zona de memoria que se utiliza por defecto
- tiempo de vida limitado por su visibilidad
  - ✓ objetos permanecen mientras sean referenciados
  - ✓ cuando dejan de ser referenciados son eliminados automáticamente por el recolector de basura
    - » el momento concreto depende de la implementación
- Esta zona de memoria es compartida por todos los *threads* de tiempo real del sistema

©2001-2002 José F. Ruiz

ImmortalMemory

dit
UPM

- Objetos creados aquí viven hasta que se acabe la aplicación
  - ✓ por ello no hay recolección de basura, la ejecución del GC nunca puede retrasar la creación de objetos en esta zona
- Una instancia de esta memoria por cada JVM
- Esta zona de memoria es compartida por todos los *threads* de tiempo real del sistema

©2001-2002 José F. Ruiz

ScopedMemory

- Tiempo de vida limitado y claramente definido
  - ✓ no hay recolección de basura
- Objetos en esta zona se eliminan cuando:
  - ✓ se sale de método *enter* o,
  - ✓ termina el último *thread* que utiliza esta zona

se eliminan aunque existan referencias a objetos en ella

- Subclases
  - ✓ *VTMemory*
    - » el tiempo de ejecución de "*new*" es variable
  - ✓ *LTMemory*
    - » el tiempo de ejecución de "*new*" es lineal

©2001-2002 José F. Ruiz

Ejemplo de *ScopedMemory*

dit UPM

```
final ScopedMemory scope =
  new LTMemory (1024);

RealtimeThread t1 = new RealtimeThread(
  null, null,
  new MemoryParameters(scope), null,
  new Runable() {
    public void run() {
      // Código del thread t1
    }
  });

RealtimeThread t2 = new RealtimeThread(
  null, null,
  new MemoryParameters(scope), null,
  new Runable() {
    public void run() {
      // Código del thread t2
    }
  });
```

```
t1.start();
t2.start();

// Esperar a que terminen t1 y t2
t1.join();
t2.join();

RealtimeThread t3 = new RealtimeThread(
  null, null,
  new MemoryParameters(scope), null,
  new Runable() {
    public void run() {
      // Código del thread t3
    }
  });

// El constructor de t3 se queda bloqueado hasta
// que se eliminan todos los objetos creados en
// scope. De esta forma es seguro que t3 arranca
// con scope limpio. Esto no pasaría con el heap.
t3.start();
```


©2001-2002 José F. Ruiz

ImmortalPhysicalMemory

dit
UPM

- Permite crear objetos dentro de un rango de memoria física, con un comportamiento específico
  - ✓ por ejemplo, mayor velocidad de acceso
- Objetos creados aquí tienen un tiempo de vida igual al de la aplicación
  - ✓ no recolección de basura
- Permite modificar parámetros de acceso a estas regiones
  - ✓ por ejemplo, parámetros de seguridad

©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- Planificación
- Manejo de memoria
- **Sincronización**
- Eventos asíncronos
- Transferencia asíncrona de control (ATC)
- Terminación de tareas
- Acceso a memoria física

©2001-2002 José F. Ruiz

Sincronización: ideas básicas

dit UPM

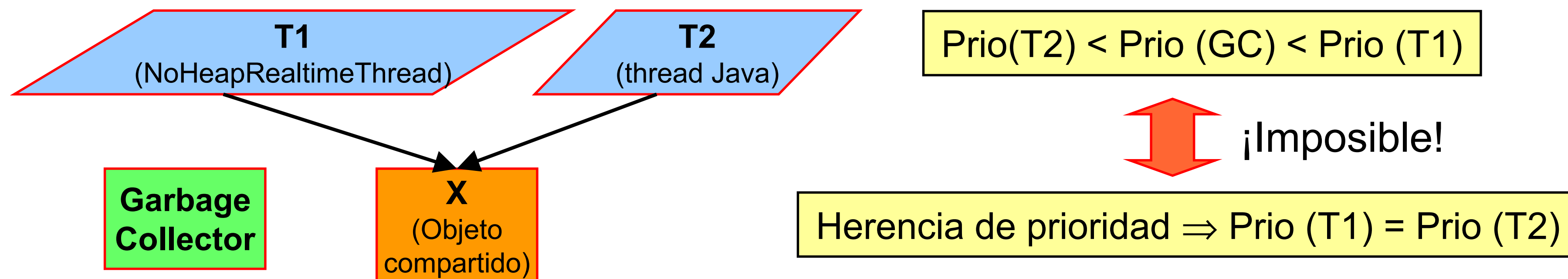
- Compatibilidad hacia atrás
  - ✓ se utiliza la palabra reservada *synchronized* de Java (monitores)
- Determinismo
  - ✓ *threads* esperando entrar en un bloque sincronizado están ordenados por prioridad
  - ✓ cuando un *thread* bloqueado pasa a estar preparado se pone como último de la cola de su prioridad
- Control de inversión de prioridad
  - ✓ el protocolo por defecto es el de herencia de prioridad (*PriorityInheritance*)
  - ✓ se pueden establecer otros algoritmos para todos o para un monitor particular, como *PriorityCeilingEmulation*

©2001-2002 José F. Ruiz

Sincronizar *threads* y *rt-threads*

dit UPM

- Problema



- Solución
  - ✓ proporcionar acceso **no bloqueante** a objetos compartidos entre los *threads* tradicionales de Java y los *NoHeapRealtimeThreads*

©2001-2002 José F. Ruiz

Wait-free queues

dit UPM

- Clases que proporcionan un conjunto de colas libres de espera
  - ✓ comunicación libre de espera entre *threads* de tiempo real y *threads* de Java
  - ✓ similar a las RT-FIFOs que comunican *threads* de Linux y *threads* de RT-Linux
- Clases definidas por la especificación
  - ✓ *WaitFreeReadQueue*
  - ✓ *WaitFreeWriteQueue*
  - ✓ *WaitFreeDequeue*

©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- Planificación
- Manejo de memoria
- Sincronización
- **Eventos asíncronos**
- Transferencia asíncrona de control (ATC)
- Terminación de tareas
- Acceso a memoria física

©2001-2002 José F. Ruiz

Eventos asíncronos

dit UPM

- Esencial en sistemas empotrados y su interacción con el entorno
  - ✓ los eventos del entorno son asíncronos a la ejecución del sistema
  - ✓ eventos asíncronos también pueden aparecer internamente
    - » desde la JVM o programables por la aplicación
- Proporciona un mecanismo 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)

©2001-2002 José F. Ruiz

Manejadores de eventos

dit UPM

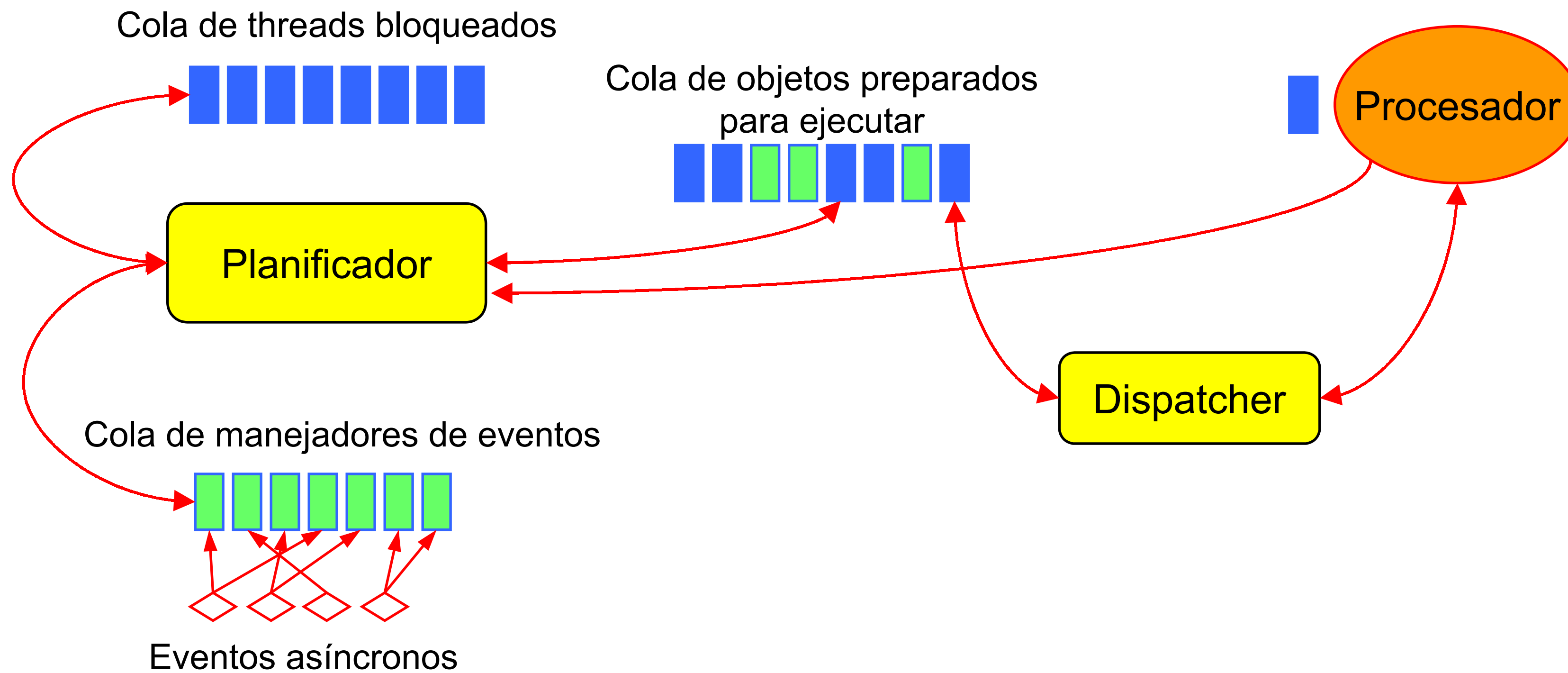
- Asociar un evento a un manejador

  AsyncEvent.addHandler(AsyncEventHandler a)

  - ✓ Una instancia de la clase *AsyncEventHandler* tiene un método llamado *handleAsyncEvent*() que se ejecuta cada vez que aparece el evento asociado
- Los eventos se disparan de dos formas
  - ✓ ejecutando el método *AsyncEvent.fire*()
  - ✓ ocurrencia del evento en el entorno
- Manejadores de eventos son objetos planificables
  - ✓ se ejecutan según sus *SchedulingParameters*
  - ✓ la ejecución de un manejador de eventos es similar a los *threads* de tiempo real

©2001-2002 José F. Ruiz

Planificación



©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- Planificación
- Manejo de memoria
- Sincronización
- Eventos asíncronos
- **Transferencia asíncrona de control (ATC)**
- Terminación de tareas
- Acceso a memoria física

©2001-2002 José F. Ruiz

ATC

dit UPM

- Funcionalidad deseada por la potencial comunidad de usarios
- Similar a las ATC´s existentes en Ada 95
- Mecanismo de las ATC´s similar a las excepciones en Java
  - ✓ sin embargo las excepciones son síncronas (el programa hace algo que causa que la excepción se eleve)
- Compatibilidad hacia atrás
  - ✓ código escrito sin conocimiento a priori de una posible interrupción no puede ser interrumpido
  - ✓ trozos de código interrumpibles deben indicarlo explícitamente

©2001-2002 José F. Ruiz

ATC (II)

dit UPM

- Elementos de la ATC
  - ✓ clase *AsynchronouslyInterruptedException* (AIE)
  - ✓ subclase *Timed*, derivada de AIE
  - ✓ interfaz *Interruptible*
  - ✓ métodos *interrupt* en *RealTimeThreads*
- Se modifica la semántica del método *interrupt*()
  - ✓ al ejecutar *t.interrupt*() se lanza una AIE al *thread* t
  - ✓ cualquier método que incluya una cláusula "*throws* AIE" ejecutará la ATC (*sleep, wait, join*, ...)
  - ✓ métodos que no incluyan la cláusula "*throws* AIE" o que estén ejecutando un método *synchronized* no se verán afectados (la ejecución de la ATC se retrasa)

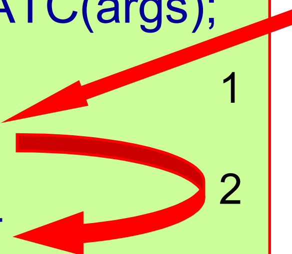
©2001-2002 José F. Ruiz

ATC (ejemplo)

dit UPM

```
public class ATC {
        public ATC (){}
        public void method1() throws AIE {// Codigo del método }
}
```

```
// RealTimeThread t1
...
ATC a = new ATC(args);
try {
  a.method1();
}
catch (AIE e) {
  // Respuesta a la
  // interrupción
}
...
```




```
// RealTimeThread t2
...
t1.interrupt();
...
```


©2001-2002 José F. Ruiz

ATC (ejemplo II)

dit UPM

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

```
LongAlg LA = new LongAlg(args);
Timed t = new
  Timed(RelativeTime(200,0));

t.doInterruptible(LA);

// El método run de LA terminará de
// ejecutarse si tarda menos de
// 200 ms. Si tarda más será
// interrumpido

t.resetTime(RelativeTime(2000,0));
t.doInterruptible(LA);

// Lo mismo que antes pero con un
// plazo de 2000 ms.
```


©2001-2002 José F. Ruiz

Mejoras en RT-Java

dit
UPM

- Planificación
- Manejo de memoria
- Sincronización
- Eventos asíncronos
- Transferencia asíncrona de control (ATC)
- **Terminación de tareas**
- Acceso a memoria física

©2001-2002 José F. Ruiz

Terminación de tareas

- Representa un requisito demandado por la comunidad de usuarios
- Se usa típicamente si el programador quiere terminar *threads* cuando aparece algún evento externo que lo requiera (cambio de modo)
- Se basa en los mecanismos de:
  - ✓ manejo de eventos asíncronos
  - ✓ transferencia asíncrona de control (ATC)
- Caracteristicas
  - ✓ Seguro y termina las tareas de modo normal
    - » Mecanismo tradicional en Java es inseguro (*stop*() y *destroy*())

©2001-2002 José F. Ruiz

Terminación de tareas (ejemplo)

dit UPM

```
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 el control
// se pasa al manejador de AIE del thread st
```

Mejoras en RT-Java

dit
UPM

- Planificación
- Manejo de memoria
- Sincronización
- Eventos asíncronos
- Transferencia asíncrona de control (ATC)
- Terminación de tareas
- **Acceso a memoria física**

©2001-2002 José F. Ruiz

Acceso a memoria física

dit
UPM

- Mecanismo esencial en sistemas empotrados
  - ✓ por ejemplo, para escribir *drivers* de dispositivos
- Dos estilos de acceso:
  - ✓ Leer y escribir directamente en memoria física
    - » *RawMemoryAccess*

      Muy útil para control de dispositivos mapeados en memoria
  - ✓ Crear objetos en memoria física
    - » *InmortalPhysicalMemory*
    - » *ScopedPhysicalMemory*

      Ejemplo, memoria *cache* manejada por el programador

©2001-2002 José F. Ruiz

Representación del tiempo

dit
UPM

- Representación en nanosegundos
  - ✓ no garantiza que el sistema operativo subyacente puede dar esa precisión
- Al menos tres clases distintas:

  HighResolutionTime (clase raiz abstracta)
  - ⤷ *AbsoluteTime*
  - ⤷ *RelativeTime*
  - ⤷ *RationalTime*

  } **añaden potencia expresiva**

    » por ejemplo, *RationalTime*(29,232) dispara una alarma 29 veces cada 232 milisegundos

©2001-2002 José F. Ruiz

Temporizadores

dit UPM

- Permite crear un temporizador que avise periódicamente o en un momento dado
- Dispara un evento cuando un temporizador expira, usando el mecanismo de eventos asíncronos
- Clase *Clock*
  - ✓ proporciona la referencia de tiempo
- Clase *Timer*
  - ✓ *PeriodicTimer* - Dispara un *AsyncEvent* periódicamente
  - ✓ *OneShotTimer* - Dispara un *AsyncEvent* en el momento requerido

©2001-2002 José F. Ruiz

Conclusiones

dit
UPM

- La propuesta de la especificación parece muy interesante y bien pensada
  - ✓ liderada por gente y empresas muy importantes, con experiencia en diversos campos
- No está claro cual de las dos propuestas saldrá ganando
  - ✓ yo apuesto por la del *Real-Time for Java Expert Group*
- Es necesaria la disponibilidad de implementaciones de calidad
  - ✓ existe una implementación de referencia hecha por TimeSys muy recientemente

©2001-2002 José F. Ruiz

Referencias

dit
UPM

- Requirements for Real-Time Extensions for the Java Platform
  - ✓ *http://www.nist.gov/rt-java*
- The Real-Time for Java Expert Group
  - ✓ *http://www.rtj.org*
  - ✓ The Real-Time for Java Expert Group, *The Real-Time Specification for Java*, 2000, Addison-Wesley.
  - ✓ La versión definitiva de su especificación salió en Noviembre de 2001
  - ✓ Hay una implementación de referencia para su especificación de RT-Java en *http://timesys.com/rtj*
- J Consortium
  - ✓ *http://www.j-consortium.org*

©2001-2002 José F. Ruiz