La Spécification Java pour le Temps Réel

ETR 2007

Serge Midonnet

Institut Gaspard-Monge, Université de Marne-La-Vallée

Nantes, 6 Septembre 2007

Outline

- Introduction
- Ordonnanceurs
  - la classe Scheduler
  - Flexibilité
  - PriorityScheduler
  - Faisabilité
- Entités Ordonnançables
  - Les RealtimeThreads
  - Les AsyncEventHandlers
- Paramètres
  - SchedulingParameters
  - ReleaseParameters
    - PeriodicParameters
    - SporadicParameters
  - Temps et Timers
  - Les ProcessingGroupParameters
  - Mémoire
  - Mémoire Immortelle et Mémoire Scope
- Détection des Fautes
  - Fautes Périodiques
  - Fautes Sporadiques
- Ressources Partagées
  - Protocoles d'Accès et Moniteurs
  - Résultats RI 1.0.1 (PIP)
  - Résultats Jtime (PIP)
- Conclusion

- Spécification de propriétés temporelles pour Java
- Historique
  - 1998: Appel à spécification: JSR-1
  - 1999: 1er Draft
  - 2002: RTSJ 1.0 approuvée par JCP
  - 2005: RTSJ 1.0.1
  - 2006: RTSJ 1.0.2

RI(Timesys), JamaïcaVM(Aicas), jRate(SourceForge), Mackinac(Sun), Apogee Aphelion JRE (LynxOs), Websphere RealTime (IBM).

- Apport du temps réel à Java:
  - choix de l'`Ordonnanceur` et plusieurs `types d'Entités Ordonnançables(EO)`;
  - `Propriétés temporelles` → `modèles d'activation`, `échéance`,
  - `plus de contrôle` → affectation de ressources, interruptions asynchrone, choix du type de mémoire.
  - `Déterminisme` → analyse de faisabilité.
- Apport de Java pour le temps réel : Simplicité et Portabilité (WORA → WOCRAM)

- un Ordonnanceur à Priorités Fixes est obligatoire $\rightarrow$ classe PriorityScheduler ,
- un minimum de 28 niveaux de priorité,
- Pri > 10 Entités Ordonnançables Prio <= 10 Threads Java
- il peut exister d'autres classes Scheduler$\rightarrow$ EDFScheduler;

- *Scheduler Scheduler.getDefaultScheduler();*
- *void Scheduler.setDefaultScheduler(Scheduler s);*
- *String Scheduler.getPolicyName();*

```
public static void main(String [] args){
      Scheduler.setDefaultScheduler(PriorityScheduler.instance());
```

Trois grandes approches pour l'intégration d'un ordonnanceur

- Le système propose un mécanisme pour insérer (plugger) un nouvel ordonnanceur → ordonnanceur pluggable, [Corba Dynamic Scheduling].
- Le système notifie l'application lors de chaque évènement nécessitant une décision d'ordonnancement, l'application décide quel thread doit s'exécuter et en informe l'ordonnanceur → ordonnanceur applicatif, [Posix Realtime extension].
- L'implantation (JVM) peut proposer differents ordonnanceurs (nécessite une modification du système (JVM) → ordonnanceur dépendant de l'implantation, [Ada 95][RTSJ].

RTSJ a adopté la troisième approche soit la moins portable. Lire [Zerzelidis & Wellings JTRES 2006] pour une approche flexible basée sur un *PriorityScheduler*.

- Ce qui est dit:
  - support de la préemption
  - support de la notion de priorité de base (fixée par l'application) et de priorité active
  - l'ordonnanceur considère la priorité active. Cette dernière est modifiable à tout moment par l'application ou par le système (synchronisations)
  - 28 niveaux de priorité au minimum (RI : 11 RTthreads <11 Threads).
  - support des PriorityInheritance et Priority Ceiling Emulation pour les objets moniteurs.
  - Une entité bloquée est placée en queue de sa file de priorité lorsqu'elle redevient éligible . Idem pour un thread appelant a méthode *yield()*.
- Ce qui n'est pas dit:
  - La position de replacement d'une entité préemptée dans sa file de priorité n'est pas spécifiée (tête de file recommandée).
  - La politique d'ordonnancement des entités dans une même file

- *le rôle essentiel de la classe Scheduler est de fournir des méthodes pour l'analyse de faisabilité*
- `admission d'une nouvelle entité dans le système` → `RealtimeThread`, `AsyncEventHandler`
- ajout ou retrait de la liste des objets ordonnançables:
  - sans condition de faisabilité
    - *boolean Scheduler.addToFeasibility();*
    - *boolean Scheduler.removeFromFeasibility();*
  - modification des paramètres sous condition de faisabilité
    - *boolean Scheduler.setIfFeasibility();*
  - pour tester la faisabilité du système
    - *boolean Scheduler.IsFeasibility();*

- on peut avoir besoin des les surcharger (analyse insuffisante, ou méthodes non synchronisées (pas obligatoire) (pb si admission dynamique).
- on peut surcharger *Scheduler* (portée tous les Scheduler) ou PriorityScheduler; Schedulable (portée tous les Schedulable (RealtimeThread, AsynEventHandler et futurs) ou une classe de Schedulable (RealtimeThreads);

- EO = instances de Classes implantant *Schedulable* (classe abstraite)

Runnable

Schedulable

Thread

RealtimeThread

NoHeapRealtimeThread

AsyncEventHandler

BoundAsyncEventHandler

```
import javax.realtime.*;
public class LaPlusSimple {
      public static void main (String [] args){
            RealtimeThread lps = new RealtimeThread(){
                  public void run () {
                        System.out.println ("hello from LaPlusSimple");
                  }
            };
            lps.start();
      }
}
```

- s'assurer que la classe lanceur est bien un RealtimeThread (modif entre les implantations RI1.0.1 et RI1.0.2).
- pour être sur: *public class LaPlusSimple extends RealtimeThread*

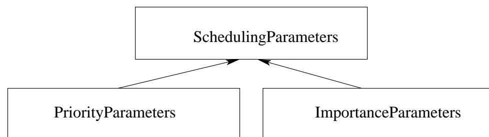
Les AsyncEventHandlers

```
AsyncEventHandler aeh=new AsyncEventHandler(){
   public void handleAsyncEvent(){
    System.out.println("hello from laPlusSimpleAperiodic");
   }
};
AsyncEvent ae = new AsyncEvent();
ae.addHandler(aeh);
ae.fire();
```

- On peut associer plusieurs AEH à un même évènement;
- On peut associer un même AEH à un plusieurs évènements ;
- Dans ce cas on en pas capable d'indentifier la source de l'évènement
- -> Surcharger AEH pour enregistrer les références des AE
- -> Ajouter une condition de déclanchement (metaEvènements)

- Paramètres des des entités ordonnançables: les classes de paramètres
  - Les paramètres d'ordonnancement: *classe SchedulingParameters*
  - Les paramètres d'activation: *classe ReleaseParameters*
  - Les paramètres de groupe: *classe ProcessingGroupParameters*
  - Les paramètres mémoire: *classe MemoryParameters*
- Classes applicables à toutes les EO

SchedulingParameters

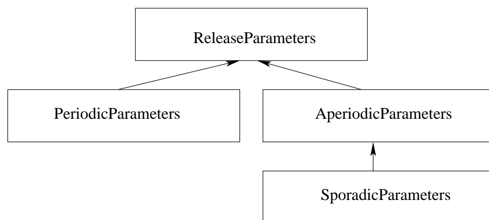



```
public class PasTropDur {
 public static void main (String [] args){
  RealtimeThread ptd = new RealtimeThread(new PriorityParameters(prio)){
    public void run () {
     System.out.println ("hello from PasTropDur");
    }};
   ptd.start();
  }}
```

```
public class launch2 extends realtimeThread{
      public launch2 (int prio){
            super(new PriorityParameters(prio));
      }
      public void run(){
            customRealtimeThread t1=new customRealtimeThread(12);
            customRealtimeThread t2=new customRealtimeThread(13);
            t1.start();
            try{
                  Thread.sleep(100);
            }catch (InterruptedException e){}
            t2.start();
            for (long indice=0; indice < 100; indice++){
                  System.out.println("Compteur: t1.c="t1.c+"-t2.c="+t2.c);
                  try{
                        Thread.sleep(30);
                  }catch(InterruptedException e){}
            }
```

```
public static void main(String [] args){
     launch2 l2 = new launch2(15);
     l2.start();
  }}
class customRealtimeThread extends RealtimeThread{
  public int c = 0;
  public customRealtimeThread(int prio){
    super(new PriorityParameters(prio));
  }
  public void run(){
    for(long k=0;k<1000000;k++){c++;} //incrémente k jusqu'à 1000000.
 }}
[root@florance]# ./tjvm -Djava.class.path=:/home/sm/coursjtr launch2
Compteur : t1.c=0 - t2.c=0
Compteur : t1.c=121628 - t2.c=0
Compteur : t1.c=254435 - t2.c=0
Compteur : t1.c=387245 - t2.c=0
Compteur : t1.c=387245 - t2.c=121628
Compteur : t1.c=387245 - t2.c=254435
Compteur : t1.c=387245 - t2.c=3737414
Compteur : t1.c=387245 - t2.c=6681064
Compteur : t1.c=387245 - t2.c=1000000
Compteur : t1.c=502966 - t2.c=1000000
```

ReleaseParameters



- éléments communs: coût; échéance relative; handlers.
- Dans la future version: BlockingTerm (pour les synchros)

```
import javax.realtime.*;
public class SimplePeriodic {
      public static void main (String [] args){
            RealtimeThread spt = new RealtimeThread(
                 new PriorityParameter(15),
                 new PeriodicParameters(
                      null,//démarrage immédiat
                      new RelativeTime(7000,0),//période
                      new RelativeTime(2000,0),//cout
                      new RelativeTime(7000,0),//échéance
                      null,//overrunHandler
                      null)//deadlineMissedHandler
                 ){
                 public void run () {
                 do {
                      System.out.println ("hello from SimplePeriodic!");
                 }while (waitForNextPeriod())
                 }
            };
            spt.start();
      }
}
```

ReleaseParameters

```
AsyncEventHandler aeh=new AsyncEventHandler(
            new PriorityParameters(20),
            new SporadicParameters(
                         new RelativeTime(4000,0), //minInterarrival
                         new RelativeTime(1000,0), //cost
                         new RelativeTime(2000,0), //deadline
                         null,
                         null);
                         )
              public void handleAsyncEvent(){
                    System.out.println("hello from laPlusSimpleSporadic");
              }
          };
AsyncEvent ae = new AsyncEvent();
ae.addHandler(aeh);
ae.fire();
```

Comparable

HighResolutionTime

AbsoluteTime

RelativeTime

RationalTime

Figure: *diagramme de classes temps*

- RationalTime prévue pour définir une fréquence a été dépréciée en version 1.0.1

```
import javax.realtime.*;
public class TOneShot{
    static boolean stop=false;
    public static void main(String [] args){
        AsyncEventHandler handler = new AsyncEventHandler() {
            public void handleAsyncEvent(){
                stop = true;
            }
        };
        OneShotTimer timer = new OneShotTimer(
            new RelativeTime(10000, 0), mise à feu dans 10 secondes
            handler);
        timer.start();
        while (!stop){
            System.out.println ("....");
          try{
                Thread.sleep(1000);
            } catch (Exception e){}
        }
        System.exit(0);
    }
}
```

```
import javax.realtime.*;
public class TPeriodique{
     public static void main(String [] args){
          AsyncEventHandler handler = new AsyncEventHandler(){
               public void handleAsyncEvent(){
                    System.out.println("hip");
               }
          };
          PeriodicTimer timer = new PeriodicTimer(
          null, // d\'emarrage imm\'ediat
          new relativeTime(1000,0)//r\'eveil toutes les secondes
          handler);

          timer.start();
       try{
              Thread.sleep(10000);//attente de 10 secondes
          }catch (Exception e){}
          timer.removeHandler(handler);
          System.out.println("hourra");
          System.exit(0);
     }
}
```

Les paramètres de groupe: *classe ProcessingGroupParameters*

- Similaire à des *ReleaseParameters* partagés par plusieurs entités schedulables.
  - un **coût** partagé correspond à un budget alloué à un ensemble d'objets Schedulable
  - une **période** correspond à la période de renouvellement du budget.
  - une **échéance** Deadline, des handlers
- Permettent de définir un serveur de tâches à un niveau logique (tâches apériodiques).
  - -> Comment choisir une politique de service des apériodique (ps,ss, ds)
  - -> A.F. les mêmes méthodes pour toutes les EO. Design de l'AF limitée.
  - -> Le mécanisme de Cost Enforcement étant optionnel les PGP sont diificilement utilisables(portables).
  - -> Création d'un nouvelle EO: classe abstraite TaskServer et classes dérivées PollingServer, DeferrableServer,

Mémoire

MemoryArea

HeapMemory
ImmortalMemory
ScopedMemory
ImmortalPhysicalMemory

VTMemory
LTMemory

VTPhysicalMemory
LTPhysicalMemory
ScopedPhysicalMemory

PhysicalMemoryManager
PhysicalMemoryTypeFilter
SizeEstimator

RawMemoryAccess
RawMemoryFloatAccess

- RTSj introduit la notion de régions

```
public abstract class MemoryArea{
protected MemoryArea(long size);
protected MemoryArea(long size, Runnable logic);
public void enter(); la mémoire est associée à l'entité ord courante pour la du
public static MemoryArea getMemoryArea (Object o);
public long memoryConsumed();
public long memoryRemaining();
public long size();
```

- **enter()**: la mémoire est associée à l'EO courante pour la durée de sa méthode logic.run(); **getMemoryArea()**: retourne la référence de la mémoire associée à l'objet o; **memoryConsumed()**: retourne le nbre de bytes utilisés dans la région; **memoryRemaining()**: retourne le nbre de bytes libres dans la région **size()**: retourne la taille de la région.

- un objet peut être affecté dans une de ces zone de manière implicite (règles):

```
public class MaClasse{
 public Object o1 = new Object();
 public static Objet o2 = new Object();
 static{
  Object o3 = new Objet();
  }
}
```

Dans l'exemple précédent, l'objet o1 est alloué dans la heap memory, les objets o2 et o3 sont alloués dans l'immortal memory.

- explicite: on utilise un objet de type *MemoryParameters* lors de la création d'une EO: toutes les allocations du RTthread ou de l'AEH sont faites dans la région choisie.
- explicite: une EO (Runnable) décide d'entrer dans une région (immortelle ou scope). Toutes les allocations sont faites dans cette région

```
ImmortalMemory.instance().enter(new Runnable){
public void run(){
// toute allocation dans run() est faite dans la mémoire immortelle
}
});
```

- Mémoire Immortelle: Les objets ne sont jamais détruits. Le programmeur doit y créer des objets réutilisables
- Mémoire Scope: Contrairement à la précédente les objets créés en mémoire scope ont une durée de vie limitée à la durée de vie de la mémoire scope .
- La mémoire Scope est détruite lorsqu'elle n'est plus utilisée: lorsqu'il n'y a plus aucun **Runnable** associé à la scope.

- Mémoire Immortelle: Les objets ne sont jamais détruits. Le programmeur doit y créer des objets réutilisables
- Mémoire Scope: Contrairement à la précédente les objets créés en mémoire scope ont une durée de vie limitée à la durée de vie de la mémoire scope .
- La mémoire Scope est détruite lorsqu'elle n'est plus utilisée: lorsqu'il n'y a plus aucun **Runnable** associé à la scope.
- LTMemory: Le temps d'allocation est linéaire en la taille de l'objet créé (hors exécution du constructeur).
- VTMemory: Le temps d'allocation est variable comme dans le tas mais plus rapide que dans le cas de la LTMemory.

- Handler de dépassement de cout: CostOverrun (AsyncEventHandler): Facultatif
- Handler de dépassement d'échéance DeadlineMiss (AsyncEventHandler): Obligatoire

- en cas de CostOverrun l'EP est immédiatement rendue inéligible.
- en cas de DeadlineMiss l'EP sera inéligible pour sa date de prochaine activation.
- Si un handler de dépassement est appelé alors l'objet ne sera rendu inéligible qu'en fin d'instance courante (*waitForNextPeriod()* sauf si le handler le rend à nouveau éligible (*reschedulePeriodic()*).
- La valeur de retour de **waitForNextPeriod()** est *true* si aucun dépassement n'a eu lieu lors de la précédente instance, *false* dans le cas contraire.

- A chaque appel de la méthode *fire()* du gestionnaire d'évènements, la tâche sporadique s'active.
- En cas de non respect de la période minimale d'un *AsyncEventHandler* plusieurs dispositions sont prévues par RTSJ.
  - en ignorer l'activation,
  - déclancher une exception,
  - le précédent évènement dans la file est remplacé,
  - l'évènement est sauvegardé.
- Ces différentes solutions sont représentées par la variable *MitViolationBehavior* (Minimum Les valeurs de comportement sont: *MitViolationIgnore*, *MitViolationExcept*, *MitViolationReplace*, *MitViolationSave*.
- L'accès à la valeur courante est possible via *getMitViolationBehavior()*, elle est modifiable grâce à la méthode *setMitViolationBehavior()*.

- Dans la conception d'un système il est réaliste d'envisager que les applications partagent des ressources et que le respect de la cohérence de ces dernières impose un accès exclusif.
- En Java la directive **synchronized** permet de déclarer que l'accès à une classe, à une méthode ou à un bloc d'instructions est contrôlé par un objet moniteur.
- Le problème typique lié à l'utilisation d'objets moniteurs est l'*inversion de priorité non bornée*.
- Le problème intervient lorsqu'une tâche est bloquée pour une durée indéterminée par une tâche de priorité inférieure avec laquelle elle ne partage aucune ressource.
- solution protocole garantissant contre les inversions de priorités non bornées dans le moniteur de l'objet partagé.

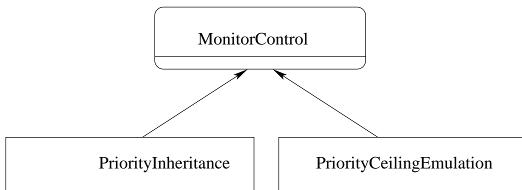


Figure: diagramme de classes Moniteurs

Blocage Direct
Blocage Indirect
demande (SC1)
accès (SC1)
libération (SC1)
t1 Priorité Haute
2 4 6 7 11
t2 Priorité Moyenne
5 11
t3 Basse Priorité
7
demande (SC1)
instant d'Héritage

exécution normale
utilisation de la ressource
inversion de priorité
utilisation de la ressource en priorité élevée

Figure: le protocole PIP

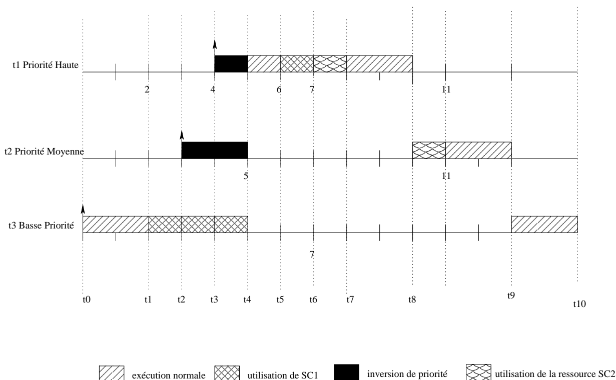
```
import javax.realtime.*;
public class Inversion extends RealtimeThread {
  public static void main(String[] args){
    SimpleDepThread t1 = new SimpleDepThread();
    t1.setName("tau1");
    SimpleThread t2 = new SimpleThread();
    t1.setName("tau2");
    SimpleDepThread t3 = new SimpleDepThread();
    t1.setName("tau3");
    t3.start();
    sleep(0.5); // on attend 500 ms pour etre sur que t3 est entr\'e en sc1
    t1.start(); // on démarre t1
    t2.start();,//puis t2

}
public class Ressource{
  public static synchronized void acces(){
    System.out.println((RealtimeThread.currentThread()).getname()+"accède à RES
    for (k=0 ; k<31120000 ; k++); //occupation pendant une seconde (à adapter)
    System.out.println((RealtimeThread.currentThread()).getname()+"libère RES1"
  }
```

```
public class SimpleThread extends RealtimeThread {
  public SimpleThread()
  public void run (){
    System.out.println("début de"+(RealtimeThread.currentThread()).getname());
    if for (k=0 ; k<31120000 ; k++); //exécution pendant une seconde
    System.out.println("fin de"+(RealtimeThread.currentThread()).getname());
  }
}
public class SimpleDepThread extends RealtimeThread {
  public SimpleDepThread()
  public void run (){
    System.out.println("début de"+(RealtimeThread.currentThread()).getname());
    Ressource.acces();
    System.out.println("fin de"+(RealtimeThread.currentThread()).getname());
  }
}
```

```
monitorControljavax.realtime.PriorityInheritance@13ee78
t3 : debut de instance 0
t3 : instance 01sec
thread3enter RES-1
t31sec
t1 : debut de instance 0
t1 : instance 01sec
t32sec
t33sec
t2 : debut de instance 0
t2 : instance 01sec
t2 : instance 02sec
t2 : instance 03sec
t2 : fin de instance 0
t34sec
t35sec
thread3leaves RES-1
thread1enter RES-1
t11sec
t12sec
t13sec
t14sec
t15sec
```

```
monitorControljavax.realtime.PriorityInheritance@7b0c1c
t3 : debut de instance 0
t3 : instance 01sec
thread3enter RES-1
t31sec
t1 : debut de instance 0
t1 : instance 01sec
t32sec
t33sec
t34sec
t35sec
thread3leaves RES-1
thread1enter RES-1
t11sec
t12sec
t13sec
t14sec
t15sec
thread1leaves RES-1
t1 : instance 01sec
t1 : fin de instance 0
t2 : debut de instance 0
t2 : instance 01sec
```

Protocoles d'Accès et Moniteurs



Object res1 = new Object(); MonitorControl policy = res1.getMonitorControlPolicy(); PriorityCeilingEmulation pce = new PriorityCeilingEmulation(20); int ceiling = pce.getDefaultCeiling(); MonitorControl.setMonitorControl(res1,pce);

- RTSJ est une avancée concrète malgré sa jeunesse
- Des améliorations attendues dans la prochaines version
  - (ex: BlockingTerm dans les ReleaseParameters)
  - Permettre d'appliquer un modèle d'activation apériodique ou sporadique à des objets de type RealtimeThread en spécifiant des méthodes *waitForNextRelease()* et *release()*).
  - transport d'inforamation entre **AsyncEvent** et **AsyncEventHAndler**
- Support des dernières versions JAva 1.5 et 1.6
- Attente forte de la version distribuée (DRTSJ)