ED2 sur Java temps réel

RTSJ (jRate)

Cours de Temps Réel Asynchrone (NFP 227)

Samia Bouzefrane & Jean-Paul Etienne

Laboratoire CEDRIC

CNAM

Exercice 1
Créer une thread temps réel (`RealtimeThread`) périodique qui s'exécute dans une mémoire scope. La thread affiche à chaque période la taille de la mémoire restante en utilisant la méthode `getCurrentMemoryArea().memoryRemaining()`. Pourquoi la thread temps réel ne s'exécute pas indéfiniment ? Expliquez.

Exercice 2
Proposer une solution au problème posé dans l'exercice précédent afin d'éviter la perte de mémoire. Il vous faudra définir dans la thread temps réel une mémoire scope et un objet `Runnable`, qui seront initialisés dans le constructeur de la thread. Ces deux éléments seront utilisés pour l'affichage à chaque période.

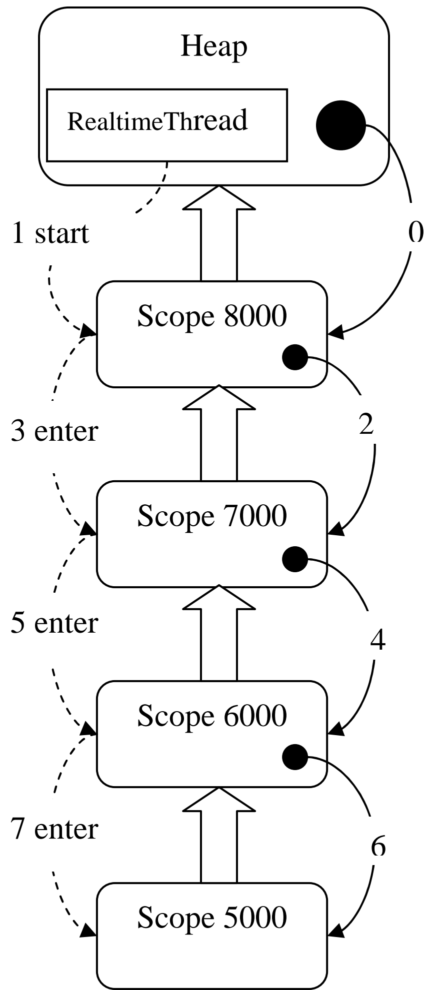
Exercice 3
Dans cet exercice nous allons créer une hiérarchie de mémoires scope, afin d'expérimenter la notion de pile de mémoires scope actives (active scope stack) en utilisant notamment les méthodes `enter`, `executeInArea`, qui nous permettront de parcourir la pile dans les deux sens.

1. Créer une thread temps réel, qui s'exécute dans un premier temps dans une mémoire scope `LT`, d'une taille de 8000 octets. Durant son exécution dans `LT`, la thread affiche la taille de `LT` en utilisant la méthode `getCurrentMemoryArea().size()`, puis la thread définit une mémoire scope de 7000 octets (`LT1`) et s'exécute dans cette dernière via la méthode `LT1.enter(..)`. Une fois dans `LT1`, la thread effectue les mêmes opérations que celles effectuées dans `LT` mais en définissant cette fois-ci une mémoire scope de 6000 octets (`LT2`), ainsi de suite jusqu'à s'exécuter dans une mémoire scope de 5000 octets (`LT3`). Une fois dans `LT3`, la thread affiche la taille de `LT3` ainsi que la longueur de la pile de mémoires (active scope stack), jusqu'ici traversées en utilisant la méthode `getMemoryAreaStackDepth()`. L'exécution de l'application est représentée par le schéma « première étape » ci-dessous.

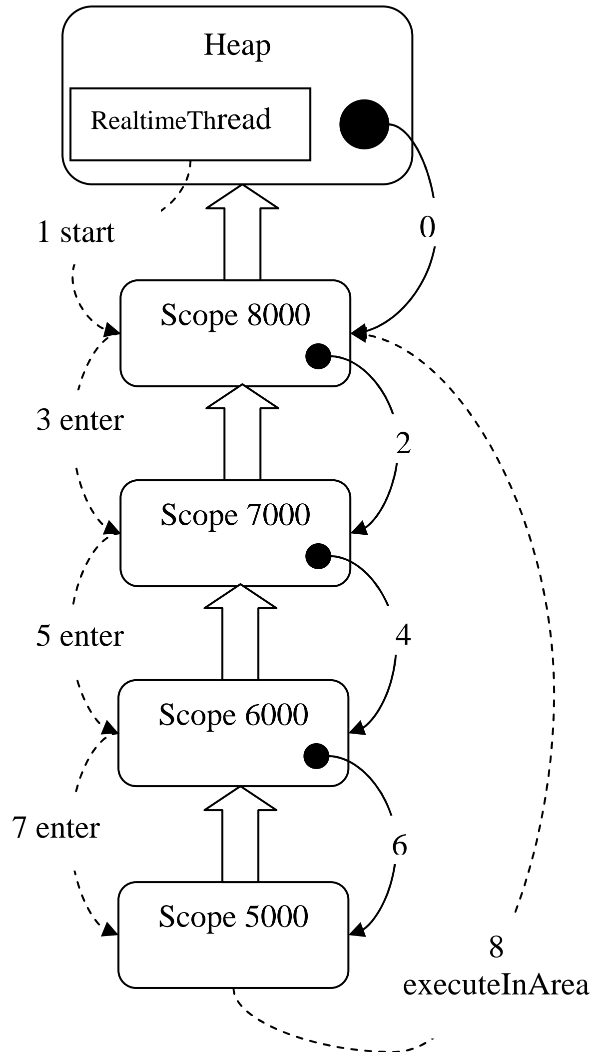
2. Une fois cette partie de l'application testée, modifier l'exécution de la thread temps réel dans la scope `LT3` en remontant d'un niveau grâce à la méthode `getOuterMemoryArea(..).executeInArea(..)`. La méthode `getOuterMemoryArea(..)` permet d'accéder aux mémoires scopes se trouvant dans la pile. La méthode `executeInArea(..)` permet d'exécuter un objet `Runnable` (dans notre cas la thread temps réel) dans une mémoire scope parent. Effectuer un `getOuterMemoryArea` en utilisant comme indice la valeur 3. Une fois la thread temps réel dans la zone mémoire indiquée, afficher la taille de la mémoire. Dans quelle mémoire scope vous vous trouvez? L'exécution de cette partie est représentée par le schéma « deuxième étape ».

3. Une fois cette deuxième partie testée, modifier l'exécution de la thread temps réel dans la dernière mémoire scope atteinte en y définissant une mémoire scope de 4000 octets. Puis exécuter la thread temps réel dans cette mémoire, via la méthode `enter`. Une fois dans cette zone, afficher la longueur de la pile de mémoires avec la méthode `getMemoryAreaStackDepth()`. Que constatez vous? (Voir le schéma « troisième étape »)

Première étape



Deuxième étape

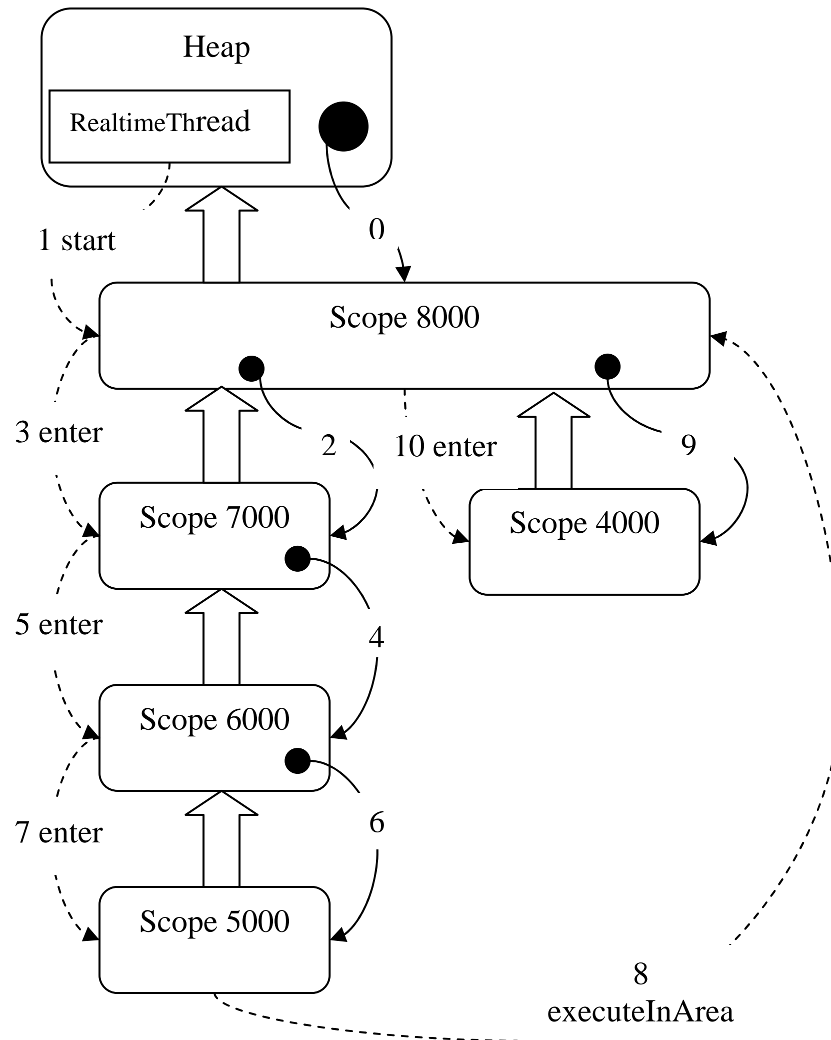


⟶ Lien d'instantiation

- - - ► Lien d'exécution

● Instance d'une scope

Troisième étape



Exercice 4

De manière générale, si une entité ordonnançable (`Runnable`, `RealtimeThread`, `NoHeapRealtimeThread`) crée lors de son exécution dans une mémoire scope (`LT`) un objet et puis poursuit son exécution dans un autre espace mémoire, elle ne pourra pas accéder à l'objet créé dans `LT` si elle revient une nouvelle fois dans `LT`, car elle n'a plus de référence vers cet objet. Pour résoudre ce problème, RTSJ offre la méthode `setPortal(..)`, qui permet de définir un objet créé dans une scope comme passerelle de cette dernière afin de pouvoir obtenir ultérieurement une référence vers l'objet, notamment à travers la méthode `getPortal()`.

L'objectif de cet exercice est de tester les mécanismes `setPortal()` et `getPortal()`. Pour cela, reprenez l'exercice 3. Modifier l'exécution de la thread temps réel dans la mémoire scope de 7000 octets (`LT1`), en créant dans un premier temps un objet de type `Integer`, puis en définissant l'objet comme passerelle de `LT1`, en utilisant la méthode `((ScopedMemory)getCurrentMemoryArea()).setPortal(objetInteger)`. Ensuite,

modifier l'exécution de la thread temps réel dans la mémoire scope de 5000 octets (`LT3`), afin qu'elle poursuive son exécution dans `LT1`. Une fois que la thread temps réel se retrouve à nouveau dans `LT1`, afficher dans un premier temps la taille de la mémoire, ensuite accéder à l'objet passerelle (dans notre cas l'objet de type `Integer`), puis afficher la valeur de l'objet. Pour accéder à la passerelle, utiliser la méthode `((ScopedMemory)getCurrentMemoryArea()).getPortal()`. Comme la méthode `getPortal` nous renvoie une entité de type `Object`, il vous vaudra faire un cast en utilisant `Integer` afin d'afficher la valeur de l'entier.

Exercice 5

Dans cet exercice nous allons voir comment on peut utiliser la notion de passerelle pour accéder à d'autres mémoires scopes. Dans l'exercice précédent, une fois que la thread temps réel se retrouve de nouveau dans `LT1`, on ne peut accéder à la mémoire scope de 6000 octets (`LT2`) qui y est définie, car on n'a pas de référence vers elle. Afin de permettre l'accès à `LT2`, modifier l'exécution de la thread temps réel `LT1` en définissant `LT2` comme la passerelle de `LT1`. Ensuite modifier l'exécution de la thread temps réel dans la partie où elle se retrouve de nouveau dans `LT1` pour qu'elle puisse accéder à `LT2` par le biais de la méthode `getPortal()`. Une fois obtenue la référence vers `LT2`, faites que la thread temps réel y poursuit son exécution (via la méthode `enter`). Une fois de nouveau dans `LT2`, afficher sa taille.

Corrigé Exercice 1

Nous constatons que la taille de la mémoire restante diminue à chaque période. Cela est dû à la méthode `System.out.println`,qui à chaque invocation, crée un nouvel objet. Cela s'explique par le fait que la mémoire scope ne va désallouer les objets créés que s'il n'existe aucune thread en activité dans son espace. A chaque période de la thread, chaque nouvel objet créé par `println` reste présent dans la mémoire scope et provoque ainsi une fuite de mémoire. Cela a pour effet l'arrêt prématuré de l'application une fois tout l'espace mémoire consommé.

```
import javax.realtime.*;

public class ScopeThread extends RealtimeThread
{

    public ScopeThread(SchedulingParameters sp, ReleaseParameters rp,
MemoryArea ma)
      {
          super(sp,rp,null,ma,null,null);
      }

    public void run()
      {
          do
          {
            System.out.println("Memory Remaining : " +
getCurrentMemoryArea().memoryRemaining());
          }
          while(waitForNextPeriod());
      }


    public static void main(String[] args)
      {
          PriorityParameters sp = new
PriorityParameters(PriorityScheduler.instance().getMaxPriority()-10);

          PeriodicParameters rp = new PeriodicParameters(
            new RelativeTime(0,0),
            new RelativeTime(1000,0),
            new RelativeTime(0,0),
            new RelativeTime(0,0),
            null,
            null);

          LTMemory LT = new LTMemory(8192,8192);

          ScopeThread s = new ScopeThread(sp,rp,LT);

          s.start();

      }
}
```

Corrigé Exercice 2

```
import javax.realtime.*;

public class ScopeThread extends RealtimeThread
{

    LTMemory tmpMem;
    Runnable tmpRun;
    long l;

    public ScopeThread(SchedulingParameters sp, ReleaseParameters rp,
MemoryArea ma)
      {
          super(sp,rp,null,ma,null,null);

          tmpMem = new LTMemory(1024,1024);

          tmpRun =
            new Runnable()
            {
                public void run()
                  {
                      System.out.println("Memory remaining : " + l);
                  }

            };
      }

    public void run()
      {
          do
          {
            l = getCurrentMemoryArea().memoryRemaining();
            tmpMem.enter(tmpRun);

          }
          while(waitForNextPeriod());
      }

    public static void main(String[] args)
      {
          PriorityParameters sp = new
PriorityParameters(PriorityScheduler.instance().getMaxPriority()-10);

          PeriodicParameters rp = new PeriodicParameters(
            new RelativeTime(0,0),
            new RelativeTime(1000,0),
            new RelativeTime(0,0),
            new RelativeTime(0,0), null, null);

          LTMemory LT = new LTMemory(8192,8192);

          ScopeThread s = new ScopeThread(sp,rp,LT);

          s.start();

      }
}
```

Corrigé Exercice 3

```
import javax.realtime.*;

public class ScopeStack extends RealtimeThread
{
     int opt=1;

     public ScopeStack(MemoryArea ma)
      {
          super(null,null,null,ma,null,null);
      }

    public void run()
      {
          switch(opt)
          {
            case 1:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT1 = new LTMemory(7000,7000);

                opt=2;

                LT1.enter(this);

                break;
            }
            case 2:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT2 = new LTMemory(6000,6000);

                opt=3;

                LT2.enter(this);

                break;
            }
            case 3:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT3 = new LTMemory(5000,5000);

                opt=4;

                LT3.enter(this);

                break;
            }
```

```
            case 4:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("StackDepth : " +
getMemoryAreaStackDepth());
                opt=5;
                getOuterMemoryArea(3).executeInArea(this);
                break;
            }
            case 5:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT4 = new LTMemory(4000,4000);

                opt=6;
                LT4.enter(this);

                break;
            }
            case 6:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("StackDepth : " +
getMemoryAreaStackDepth());

                break;
            }
          }
      }

    public static void main(String[] args)
      {
          LTMemory LT = new LTMemory(8000,8000);

          ScopeStack ss = new ScopeStack(LT);
          ss.start();

          try
          {
            LT.join();
          }
          catch(Exception ie)
          {
            ie.printStackTrace();
          }

          System.out.println("finish!!");

      }
}
```

Corrigé Exercice 4

```
import javax.realtime.*;

public class ScopeStack extends RealtimeThread
{
     int opt=1;

     public ScopeStack(MemoryArea ma)
      {
          super(null,null,null,ma,null,null);
      }

    public void run()
      {
          switch(opt)
          {
            case 1:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT1 = new LTMemory(7000,7000);

                opt=2;

                LT1.enter(this);

                break;
            }
            case 2:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                Integer i = new Integer(7);

                ((ScopedMemory)getCurrentMemoryArea()).setPortal(i);

                LTMemory LT2 = new LTMemory(6000,6000);

                opt=3;

                LT2.enter(this);

                break;
            }
            case 3:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT3 = new LTMemory(5000,5000);

                opt=4;

                LT3.enter(this);

                break;
            }
```

```
            case 4:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("StackDepth : " +
getMemoryAreaStackDepth());

                opt=5;
                getOuterMemoryArea(2).executeInArea(this);
                break;
            }
            case 5:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("get Integer from Scope 7000 : " +
((Integer)((ScopedMemory)getCurrentMemoryArea()).getPortal()).toString());
            }

          }


      }

    public static void main(String[] args)
      {
          LTMemory LT = new LTMemory(8000,8000);

          ScopeStack ss = new ScopeStack(LT);
          ss.start();

          try
          {
            LT.join();
          }
          catch(Exception ie)
          {
            ie.printStackTrace();
          }

          System.out.println("finish!!");

      }
}
```

Corrigé Exercice 5

```
import javax.realtime.*;

public class ScopeStack extends RealtimeThread
{
     int opt=1;

     public ScopeStack(MemoryArea ma)
      {
          super(null,null,null,ma,null,null);
      }

    public void run()
      {

          switch(opt)
          {
            case 1:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT1 = new LTMemory(7000,7000);

                opt=2;

                LT1.enter(this);

                break;
            }
            case 2:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT2 = new LTMemory(6000,6000);

                ((ScopedMemory)getCurrentMemoryArea()).setPortal(LT2);

                opt=3;

                LT2.enter(this);

                break;
            }
            case 3:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                LTMemory LT3 = new LTMemory(5000,5000);

                opt=4;

                LT3.enter(this);

                break;
            }
```

```
            case 4:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("StackDepth : " +
getMemoryAreaStackDepth());

                opt=5;
                getOuterMemoryArea(2).executeInArea(this);
                break;

            }
            case 5:
            {

                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                System.out.println("get PortalScope from Scope 7000 : " +
((ScopedMemory)((ScopedMemory)getCurrentMemoryArea()).getPortal()).size());

                opt=6;

((ScopedMemory)((ScopedMemory)getCurrentMemoryArea()).getPortal()).enter(this);

                break;
            }
            case 6:
            {
                System.out.println("Memory Area : " +
getCurrentMemoryArea().size());

                break;
            }
          }
      }

    public static void main(String[] args)
      {
          LTMemory LT = new LTMemory(8000,8000);

          ScopeStack ss = new ScopeStack(LT);
          ss.start();

          try
          {
            LT.join();
          }
          catch(Exception ie)
          {
            ie.printStackTrace();
          }

          System.out.println("finish!!");
      }
}
```