INF5170 — Programmation parallèle
Examen final (Automne 2008)

Durée : 13h30 – 16h30 (3 heures) **Documentation autorisée :** Toute documentation personnelle.

Remarques :

- L'examen comporte cinq (5) questions, comptant respectivement pour 10, 10, 15, 10 et 10 points, pour un total sur 55.

1. Sémaphores (MPD) (10 pts)

Le programme MPD 1 présente un petit exemple d'utilisation des sémaphores en MPD.

```
int x = 2;

sem s1 = 0,
    s2 = 1;

process p1 { P(s2); x = 3 * x; V(s2); V(s1); }

process p2 { P(s2); P(s1); x = 2 * x; V(s1); V(s2); }

process p3 { P(s1); P(s2); x = x + 5; V(s1); V(s2); }

final      { printf( "x = %d\n", x ); }
```

Programme MPD 1: Programme MPD avec `process` et sémaphores.

a. Pour chacun des sémaphores, indiquez de quel type de sémaphore il s'agit (binaire/généralisé, exclusion mutuelle/signalisation?).

b. Dans ce programme, il y a possibilité de *deadlock*. Indiquez clairement *dans quelle situation*. Indiquez aussi quelles sont les sorties qui peuvent être imprimées par ce programme dans les autres exécutions sans *deadlock*?

c. Serait-il possible d'éliminer le *deadlock*? Si oui, comment et quelles seraient alors les sorties possibles qui seraient imprimées?

2. Classe de la bibliothèque `java.util.concurrent` (Java) (10 pts)

La classe présentée dans le programme Java 1 est une mise en oeuvre (partielle) des principales opérations d'une des classes de la bibliothèque `java.util.concurrent` décrites dans l'appendice A du document «La programmation concurrente en Java».

a. Identifiez cette classe et chacune des méthodes `m0`, `m1` et `m2` et décrivez brièvement dans vos propres mots à quoi servent les objets d'une telle classe.

b. Est-ce que ce serait correct, dans `m2`, de supprimer l'appel à `synchronized( c )`? Justifiez brièvement votre réponse.

```
public class Mystere {
    private AtomicInteger c;

    Mystere( int c ) {
       this.c = new AtomicInteger( c );
    }

    public int m0() {
       return( c.get() );
    }

    public void m1() {
       synchronized( c ) {
           if ( c.get() > 0 ) {
              try { c.wait(); } catch( InterruptedException ie ) {}
           }
       }
    }

    public void m2() {
       synchronized( c ) {
           c.decrementAndGet();
           if ( c.get() > 0 ) {
              try { c.wait(); } catch( InterruptedException ie ) {}
           } else {
              c.notifyAll();
           }
       }
    }

    public String toString() {
      ...
    }
}
```

Programme Java 1: Classe `Mystere.java` à identifier.

3. Mise en oeuvre d'une `IStructure` (Java) (15 pts)

Une I-structure est une forme de *tableau* composé d'un certain nombre de cellules — la taille est spécifiée au moment de l'allocation de la `IStructure` — où chaque cellule assure une *synchronisation implicite* entre producteur (unique) et consommateurs (multiples) de cette cellule. Les cellules d'une `IStructure` possèdent donc le comportement suivant :

- Initialement, toutes les cellules sont *vides*.
- Une cellule d'une `IStructure` peut être lue avec `get`. Toutefois, lorsque la cellule est vide, l'appelant *est mis en attente* jusqu'à ce que la cellule ait été remplie par un autre *thread*.
- Une cellule d'une `IStructure` ne peut être écrite, avec `put`, qu'une seule et unique fois. C'est une *erreur* (`AlreadyFullException`) d'écrire plus d'une fois dans une cellule donnée d'une `IStructure`.

Le programme Java 2 définit les opérations devant être supportées par une `IStructure` — pour simplifier, on suppose que l'index `i` reçu en argument est toujours valide (i.e., `i >= 0` et `i <` taille de la `IStructure`) :

```
interface IStructure<T> {
    void put( int i, T v ) throws AlreadyFullException;

    T get( int i );
}

class AlreadyFullException extends RuntimeException{}
```

Programme Java 2: Interface pour `IStructure`.

[5] a) Des exemples d'utilisation de `IStructure`s sont présentés dans le Programme Java 3 (page suivante, que vous pouvez déchirer pour la consulter plus facilement).

Soit alors le cas de test `exemple_final()` :

a. Quelle doit-être la valeur de `RESULTAT` pour que ce test fonctionne correctement?

b. De façon plus générale, pour un `N` arbitraire, que permet de calculer le code de ce test? Est-ce que l'approche «classique» pour résoudre ce même problème serait plus efficace? Justifiez brièvement votre réponse.

[10] b) Définissez une classe Java `IStructureMonitor` qui met en oeuvre cette interface, avec un constructeur approprié qui reçoit en argument la taille (le nombre de cellules) de la `IStructure`.

Contrainte importante de mise en oeuvre : Vous devez éviter de verrouiller l'ensemble de la `IStructure` pour accéder à une des cellules. En d'autres mots, votre solution ne devrait verrouiller que la cellule requise, sans empêcher d'autres *threads* d'accéder aux autres cellules.

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class TesterIS {

    @Test public void de_base_ok() {
       IStructure<Integer> is = new IStructureMonitor<Integer>( 4 );

       is.put( 1, 10 );
       is.put( 3, 30 );

       assertEquals( (Integer) 10, is.get(1) );
       assertEquals( (Integer) 30, is.get(3) );
    }

    @Test public void deux_threads_put_avant_get_ok() {
       final IStructure<Integer> is = new IStructureMonitor<Integer>( 4 );

       new Thread( new Runnable() { public void run() { is.put( 1, 10 ); } } ).start();
       new Thread( new Runnable() { public void run() { is.put( 3, 30 ); } } ).start();

       assertEquals( (Integer) 10, is.get(1) );
       assertEquals( (Integer) 30, is.get(3) );
    }

    @Test(expected=AlreadyFullException.class) public void de_base_pasok() {
       IStructure<Integer> is = new IStructureMonitor<Integer>( 2 );
       is.put( 1, 10 );
       is.put( 1, 30 );
    }

    // -------------------------------------------------------

    final int RESULTAT = ?;  // Valeur a determiner (question a)

    @Test public void exemple_final() {
       final int N = 10;
       final IStructure<Integer> is = new IStructureMonitor<Integer>( N );
       is.put(0, 0);
       is.put(1, 1);
       for( int i = 2; i < N; i++ ) {
           final int k = i;
           new Thread( new Runnable() {
                  public void run() {
                     is.put( k, is.get(k-1) + is.get(k-2) );
                  } } ).start();
       }
       assertEquals( (Integer) RESULTAT, is.get(N-1) );
    }
}
```

Programme Java 3: Exemples d'utilisation d'une `IStructure`.

4. Mise en oeuvre parallèle de l'opération `map` (Java) (10 pts)

On a vu en cours comment une approche «*Map/Reduce*» pouvait être réalisée en Java, et ce pour une classe `MapReduceArray` permettant de manipuler des collections d'éléments définies avec un tableau simple. Pour ce faire, il faut tout d'abord définir une interface appropriée pour représenter la fonction à appliquer :

```
interface Fonction<T,R> {
    R appliquer( T e );
}
```

Le Programme Java 4 présente la solution séquentielle présentée en cours.

Donnez l'extrait de code Java qui permettrait de mettre en oeuvre, de façon parallèle, la procédure `map` (et uniquement cette procédure).

Vous n'avez pas besoin de réécrire l'en-tête de la fonction, l'instruction d'allocation du tableau `r` ou l'instruction `return` ; il vous suffit uniquement de donner le bout de code qui remplacerait la boucle `for`.

Contrainte de mise en oeuvre : Cette mise en oeuvre parallèle doit être une version à *granularité (très!) fine* — donc pas besoin de travailler *avec des tranches*.

```
import java.util.concurrent.*;

public class MapReduceArray<T> {
    private T[] elems;

    MapReduceArray( T elems[] ) {
       this.elems = elems;
    }


   @SuppressWarnings("unchecked")
   <R> MapReduceArray<R> map( Class<?> klass,  Fonction<T,R> f ) {
       R[] r = (R[]) java.lang.reflect.Array.newInstance(klass, elems.length);

       for ( int i = 0; i < elems.length; i++ ) {
           r[i] = f.appliquer( elems[i] );
       }

       return new MapReduceArray( r );
    }


    T reduce( T elemNeutre, OpBinaire<T> op ) {
       T res = elemNeutre;
       for ( int i = 0; i < elems.length; i++ ) {
           res = op.evaluer( elems[i], res );
       }
       return res;
    }

    T[] toArray() {
       return elems;
    }
}
```

Programme Java 4: Mise en oeuvre séquentielle pour une classe `MapReduceArray`.

5. Opération collective de réduction (MPD ou Java) (10 pts)

Un modèle de programmation parallèle souvent utilisé est le modèle SPMD (*Single Program Multiple Data*). Dans ce modèle, les *threads* sont généralement de granularité assez grossière et leur nombre est généralement le même que le nombre de processeurs.

Les langages fondés sur le modèle SPMD fournissent de nombreuses opérations de communication et synchronisation *collectives*, lesquelles jouent deux rôles :

- Elles servent de point de synchronisation : un *thread* qui exécute l'opération bloque jusqu'à ce que tous les autres *threads* soient prêts à exécuter cette même opération.
- Elles servent aux divers *threads à échanger de l'information.*

Une opération typique est `broadcast`, terme qui se traduit en français par «diffuser». Dans cette opération, l'un des *threads*, le `diffuseur`, cherche à transmettre une valeur à chacun des autres `threads`.

L'interface pour une telle opération de diffusion peut être représentée par une opération exportée par une ressource MPD, telle qu'indiquée dans le programme MPD 2.

```
resource Broadcast
  op broadcast( ref int elem, int numThread, int diffuseur )
body Broadcast( int nbThreads )
...
```

Programme MPD 2: Interface MPD pour une opération `broadcast`.

Plus précisément, cette ressource/opération s'utilise de la façon suivante :

- L'argument `nbThreads` du constructeur indique le nombre de *threads* qui (toujours!) seront impliqués dans l'utilisation de l'opération `broadcast`.
- Un appel à `broadcast` se fait en spécifiant trois (3) arguments :
  - `elem` : L'élément à diffuser.
  - `numThread` : Le numéro du *thread* appelant — on suppose que ces numéros vont de 1 à `nbThreads` et que *i*) chaque *thread* possède un numéro unique ; *ii*) chaque *thread* spécifie correctement son numéro et participe à l'opération de synchronisation (donc effectue un appel approprié à `broadcast`).
  - `diffuseur` : Le numéro du *thread* qui doit diffuser (transmettre) sa valeur aux autres *threads*. On suppose que tous les *threads* spécifient correctement la même valeur pour cet argument.
- Tant que tous les *threads* n'ont pas effectué leur appel à `broadcast`, aucun résultat n'est produit/retourné. Donc, les `nbThreads-1` premiers *threads* qui appellent `broadcast` bloquent, en attente du résultat.
- Lorsque les `nbThreads` *threads* ont tous effectué leur appel à `broadcast`, la valeur `elem` spécifiée par le *thread* `diffuseur` est retournée à chacun des *threads*, lesquels poursuivent ensuite leur exécution.

```
import Broadcast;

printf( "Resultat pour diffuseur = 2\n" );

cap Broadcast b = create Broadcast( 3 );

process p [k = 1 to 3] {
  int a = 10 * k;
  printf( "Thread %d (avant): a = %d\n", k, a );
  b.broadcast( a, k, 2 )
  printf( "Thread %d (apres): a = %d\n", k, a );
}
```

Programme MPD 3: Un petit programme utilisant le module de diffusion d'une valeur.

Le programme MPD 3 présente un extrait d'un petit programme utilisant cette opération. Le résultat produit par l'exécution de ce programme est présenté à la Figure 1.

Figure 1 Trace d'exécution pour le programme MPD 3.

```
$ tester-broadcast
Resultat pour diffuseur = 2
Thread 1 (avant): a = 10
Thread 2 (avant): a = 20
Thread 3 (avant): a = 30
Thread 3 (apres): a = 20
Thread 1 (apres): a = 20
Thread 2 (apres): a = 20
```

Écrivez, dans la notation de «votre choix» (ou presque : pseudo-MPD, MPD ou Java), un *moniteur* permettant de réaliser une telle opération `broadcast`.