INF5171 — Programmation concurrente et parallèle
Examen final (Hiver 2011)

Durée: 18h00 – 21h00 **Documentation :** Documentation personnelle autorisée.

Nom: _____

Code permanent: | | | | | | | | | | | | |

1	2	3	4	5	Total
/15	/10	/10	/10	/10	/55
XXX		XXX			

- L'examen comporte cinq (5) questions. Certaines d'entre elles comportent des points bonus (maximum 10 points).
- Pour les problèmes demandant d'écrire du code MPD, vous devez utiliser une notation semblable à MPD, mais pas nécessairement identique. Entre autres, dans le cas des instructions `co`, vous pouvez utiliser une notation *moins stricte* que la notation acceptée par le compilateur. Par exemple, le compilateur MPD *n'accepterait pas* l'instruction `co` suivante, parce que le corps de l'instruction `co` n'est pas une invocation :

```
co [i = 1 to n]
  x[i] = y[i] + z[i];
oc
```

  Dans le cadre de l'examen, *cette notation (*sans procédure auxiliaire*) sera acceptée* — en fait, *n'importe quelle* séquence de code dans un `co` sera acceptée, y compris des instructions `for` ou `if`. Toutefois, n'abusez pas de cette possibilité et, si cela vous semble utile, vous pouvez évidemment introduire une ou des procédures auxiliaires. (Pas besoin non plus de respecter l'ordre strict des déclarations — déclaration d'une procédure avant sa première utilisation.)
- Si nécessaire, vous pouvez utiliser, *sans les définir*, les procédures suivantes :

```
# Fonctions pour le calcul des bornes d'une tranche.
procedure inf( int i, int n, int nbProcs ) returns int r
procedure sup( int i, int n, int nbProcs ) returns int r

# Operations pour l'initialisation et manipulation du sac de taches.
procedure initSacDeTaches( int bi, int bs )
procedure obtenirTache( res int i, res int j ) returns bool disponible
```

1. Interactions entre planètes (pseudo-MPD) (15 pts)

Le programme MPD 1 présente une version pseudo-MPD du calcul de mouvement des planètes (devoir 2). La procédure `fetchAndAddVecteur` permet d'incrémenter, de façon atomique, une référence à un `Vecteur`. Plus précisément, on suppose que des appels concurrents à `fetchAndAddVecteur` qui s'exécutent **sur des références (pointeurs) distinctes** pourront s'exécuter de façon parallèle, sans interférence. Par contre, des appels concurrents s'effectuant sur le même pointeur s'exécuteront de façon mutuellement exclusive.

Remarque : Le passage par référence dans `calculerForces` et `deplacerPlanetes` est utilisé pour des raisons de performance (i.e., pour éviter de copier les tableaux), à l'exception de l'argument `forces` de `calculerForces` qui, lui, est effectivement modifié.

a. Donnez le code de `calculerForces` en respectant les contraintes suivantes :

- Utilisez une distribution *cyclique* du travail entre les *threads*.
- Utilisez `fetchAndAddVecteur` pour cumuler les différentes forces sur une planète.
- Évitez de calculer deux fois les forces entre deux planètes ⇒ utilisez la propriété de *symétrie* = la force de P_1 sur P_2 est l'inverse de P_2 sur P_1.
- L'opérateur «`@`» retourne l'adresse d'une variable (valeur de type `ptr`), par ex. : `int x` ⇒ `@x` est de type `ptr int`, `T a[*]` ⇒ `@a[i]` est de type `ptr T`, etc.

Écrivez le code pour votre solution dans l'espace prévu à cette fin à la page 4.

b. Est-ce que cette solution est de type SPMD? Justifiez brièvement votre réponse.

c. Est-ce que cette solution serait une bonne solution lorsque le nombre de *threads* est très grand? Justifiez brièvement votre réponse.

```
procedure fetchAndAddVecteur( ptr Vecteur x, Vecteur incr ) returns Vecteur r
{
  < r = x^; x^ = plus( x^, incr ) >
}



procedure calculerForces( ref Planete planetes[*],
                          int numThread,
                          int nbThreads,
                          ref Vecteur forces[*] )
{
  // Code a completer: voir page suivante.
}



procedure deplacerPlanetes( ref Planete planetes[*],
                            int numThread,
                            int nbThreads,
                            real periode,
                            ref Vecteur forces[*] )
{
  for [i = numThread to ub(planetes) by nbThreads] {
    deplacer( planetes[i], forces[i], periode )
  }
}



procedure simulerPAR( Systeme sys, int nbIterations, real periode )
{
  int nbPlanetes = sys^.nbPlanetes;
  int nbThreads = min( getenvint( "MPDPARALLEL", 1 ), nbPlanetes );

  cap Barriere b = create Barriere( nbThreads );

  for [it = 1 to nbIterations] {
    Vecteur forces[nbPlanetes] = ([nbPlanetes] vecteur(0,0));

    co [numThread = 1 to nbThreads]
      calculerForces( sys^.planetes^, numThread, nbThreads, forces );
      b.attendre();
      deplacerPlanetes( sys^.planetes^, numThread, nbThreads, periode, forces );
    oc
  }
}
```

Programme MPD 1: Méthode parallèle de simulation des planètes.

```
  procedure calculerForces( ref Planete planetes[*],
                            int numThread,
                            int nbThreads,
                            ref Vecteur forces[*] )
  {









}
```

2. Sémaphores (MPD) (10 pts)

Soit le programme MPD suivant, qui utilise des sémaphores.

```
int x = 10;
sem s1 = 1,
    s2 = 0;

process p1 { P(s2); P(s1); x = x + 5; V(s1); }

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

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

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

a. Pour chacun des sémaphores `s1` et `s2`, indiquez de quel type de sémaphore il s'agit (binaire/généralisé, exclusion mutuelle/signalisation?).

b. Indiquez quelles sont les sorties qui peuvent être imprimées par ce programme?

c. Dans ce programme, il y a possibilité d'interblocage (*deadlock*). Indiquez dans quelle(s) situation(s) un tel blocage peut survenir. Serait-il possible de modifier *légèrement* le code de façon à éliminer ce(s) blocage(s)? Si oui, de quelle façon?

Bonus (5 pts) Réécrivez le code MPD de façon à ce qu'il ait un comportement équivalent, et ce **sans utiliser de sémaphores**, et en remplaçant `final` par `process p4` ainsi qu'en utilisant des instructions `< await(...) ...>`. Si nécessaire, vous pouvez introduire des variables auxiliaires *autres que des sémaphores*. Revoici le code du programme initial (pour éviter de retourner à la page précédente) :

```
int x = 10;
sem s1 = 1, s2 = 0;

process p1 { P(s2); P(s1); x = x + 5; V(s1); }
process p2 { P(s1); P(s2); x = 2 * x; V(s1); }
process p3 { P(s1); x = x - 5; V(s1); V(s2); V(s2); }

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

3. Classe `Mystere` (Java) (10 pts)

Soit l'interface et l'objet Java suivants :

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

final Fonction<Integer,Boolean> estPair = new Fonction<Integer,Boolean>() {
    public Boolean appliquer( Integer i ) { return i % 2 == 0; }
};
```

Soit la classe Java 1 (p. 8) qui utilise des fonctionnalités de `java.util.concurrent`.

a. Expliquez brièvement ce que fait cette classe — **quoi?**

Donnez aussi un exemple d'utilisation sous la forme suivante (assertion JUnit) :

```
... Code pour définir résultatObtenu à l'aide de la classe Mystere ...
assertEquals( résultatAttendu , résultatObtenu );
```

b. Expliquez brièvement de quelle façon, en termes de stratégie de programmation parallèle, cette classe fonctionne — **comment?**

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

public class Mystere<T> implements Callable<Integer> {
    final private T elems[];
    final private Fonction<T,Boolean> fct;
    final private int nbt;
    final private int tt;

    Mystere( T elems[], Fonction<T,Boolean> fct, int nbt, int tt ) {
        this.elems = elems;
        this.nbt = nbt;
        this.fct = fct;
        this.tt = tt;
    }

    @SuppressWarnings("unchecked")
    public Integer call() {
        assert elems.length % tt == 0;

        int nn =  elems.length / tt;

        final ExecutorService pool = Executors.newFixedThreadPool( nbt );

        Future<Integer> res[] = (Future<Integer>[]) new Future[nn];
        for ( int i = 0; i < nn; i++ ) {
            final int bi = i * tt;
            final int bs = (i+1) * tt - 1;
            res[i] = pool.submit( new Callable<Integer>() {
                public Integer call() {
                    int r = 0;
                    for ( int i = bi; i <= bs; i++ ) {
                        if ( fct.appliquer( elems[i] ) ) {
                            r += 1;
                        }
                    }
                    return r;
                } } );
        }

        int rr = 0;
        try {
            for ( int i = 0; i < nn; i++ ) {
                rr += res[i].get();
            }
        } catch( Exception e ){}
        return rr;
    }
}
```

Classe Java 1: Une classe Java «`Mystere`».

4. `MCellule` (Java) (10 pts)

Une `MCellule` est une structure de données parallèle qui permet une synchronisation implicite entre producteurs et consommateurs. Le comportement d'une `MCellule` est le suivant :

- La valeur initiale est spécifiée au moment où la `MCellule` est créée (constructeur).
- Une `MCellule` ne peut être lue que lorsqu'elle est *pleine*. La lecture du contenu d'une `MCellule` a aussi pour effet **de vider la `MCellule` de son contenu**. La lecture d'une `MCellule` vide a pour effet de mettre le lecteur en attente jusqu'à ce qu'une écriture soit effectuée.
- Une `MCellule` peut être écrite plusieurs fois, mais pour qu'une écriture soit valide, la `MCellule` **doit être vide**. C'est une *erreur* d'écrire dans une `MCellule` pleine.

  L'état d'une `MCellule` alterne donc entre *pleine*, *vide*, *pleine*, *vide*, etc.

```
interface MCellule<T> {
    void ecrire( T v ) throws DejaPleineException;

    T prendre();
}

class DejaPleineException extends RuntimeException{}
```

Interface Java 1: Interface pour une `MCellule`.

L'interface Java 1 présente un type `MCellule` qui décrit la signature des deux opérations clés de ce type. Un exemple d'utilisation est présenté dans le Programme Java 1.

Une classe Java `MCelluleMonitor`, partiellement définie, qui met en oeuvre l'interface `MCellule` (avec un constructeur qui reçoit en argument la valeur initiale de la `MCellule`) est présentée à la page 11. Complétez les méthodes `ecrire` et `prendre`.

Bonus (2 pts) Donnez un nom plus significatif pour la méthode `mystere` du programme Java 1.

```
private Integer mystere( MCellule<Integer> mc, Integer n ) {
    Integer res = mc.prendre();
    mc.ecrire( res + n );
    return res;
}


@Test public void plusieurs_threads_prendre_avant_ecrire_ok() {
    final int NB = 100;
    final MCellule<Integer> mc = new MCelluleMonitor<Integer>( 0 );

    Thread t[] = new Thread[NB];
    for ( int i = 0; i < NB; i++ ) {
        final int ifinal = i;
        t[i] = new Thread( new Runnable() {
                public void run() { mystere( mc, ifinal ); }
            } );
        t[i].start();
    }

    for ( int i = 0; i < NB; i++ ) {
        try { t[i].join(); } catch( InterruptedException e ) {};
    }

    assertEquals( (Integer)  (NB * (NB-1) / 2), mc.prendre() );
}
```

Programme Java 1: Exemple d'utilisation d'une `MCellule` (extrait d'un programme de tests JUnit).

```
public class MCelluleMonitor<T> implements MCellule<T> {

    T val;
    boolean pleine;


    public MCelluleMonitor( T valInitiale ) {
      this.val = valInitiale;
      this.pleine = true;
    }

   synchronized public void ecrire( T v ) {









    }


    synchronized public T prendre() {









    }
}
```

5. Opération de dispersion d'un tableau (pseudo-MPD) (10 pts)

Les langages fondés sur le modèle SPMD (*Single Program Multiple Data*) fournissent de nombreuses opérations *collectives* de communication et synchronisation. Ces instructions...

- servent de point de synchronisation — un *thread* qui exécute l'opération bloque jusqu'à ce que tous les autres *threads* aient aussi exécuté cette opération ;
- servent aux divers *threads à échanger de l'information.*

Une opération typique est `scatter` — qu'on peut traduire par «disperser» — où l'un des *threads*, la `source`, cherche à disperser le contenu d'un tableau entre divers `threads`.

Ici, pour simplifier, on va supposer que la taille du tableau est égale au nombre de *threads* (y compris la source). Donc, après l'appel, chaque *thread* aura reçu un unique élément. On va aussi supposer que chaque *thread* est identifié par un numéro entre 1 et `nbThreads` et que les bornes du tableau à disperser sont aussi comprises entre ces bornes. Après l'appel à `scatter`, chaque *thread* aura donc reçu l'élément du tableau correspondant à son index/numéro.

L'interface d'une telle opération `scatter` associée à un **moniteur** est donc la suivante:

```
monitor Scatter
  op scatter( int numThread, int threadSource, ptr [*]int a ) returns int elem;
body Scatter( int nbThreads )
```

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

- L'argument `nbThreads` du constructeur indique le nombre de *threads* qui seront impliqués (à chacun des appels) dans l'utilisation de l'opération `scatter`.
- Un appel à `scatter` se fait en spécifiant trois (3) arguments :
  - `numThread` : Le numéro du *thread* appelant.
  - `source` : Le numéro du *thread* qui doit disperser le contenu de son tableau aux autres *threads.*
  - `a` : Un pointeur vers le tableau que la `source` veut disperser. Les *threads* autres que `source` peuvent (devraient) simplement utiliser la valeur `null`.
- Tant que tous les *threads* n'ont pas effectué l'appel à `scatter`, aucun résultat n'est retourné. Donc, les `nbThreads-1` premiers *threads* qui appellent `scatter` bloquent.
- Lorsque les `nbThreads` *threads* ont effectué leur appel à `scatter`, la valeur `elem` appropriée est retournée au *thread* appelant, lequel poursuit ensuite son exécution.

Le programme MPD 2 présente un exemple d'utilisation de cette opération. Un résultat possible produit par l'exécution de ce programme pour `N=5` et `source=2` est présenté à la Figure 1.

Écrivez (pseudo-MPD) un **moniteur** permettant de réaliser une telle opération `scatter` (avec la propriété *implicite* d'exclusion mutuelle et des *variables de condition*).

Attention : Pour que le comportement soit toujours correct, il faut attendre d'avoir affecté le bon résultat (à `r`) avant de permettre une autre vague d'utilisation.

Bonus (3 pts) : Un bonus sera accordé si une erreur d'exécution (par ex., avec un `assert`) est signalée si tous les *threads* n'indiquent pas la même valeur pour `source`.

```
int N; getarg(1, N);
int source; getarg(2, source);

cap Scatter b = create Scatter( N );

process p [k = 1 to N] {
  int r;
  if (k == source) {
    # La source definit et indique le tableau a distribuer.
    int a[N]; for [i = 1 to N] { a[i] = 100 * i + k; }
    r = b.scatter( k, source, @a );
  } else {
    # Les autres threads peuvent mettre null.
    r = b.scatter( k, source, null );
  }
  printf( "Thread %2d: r = %4d\n", k, r );
  assert( r == 100*k+source, "Pas bon resultat" );
}

# Rappel: L'operateur '@' permet d'obtenir l'adresse d'une variable,
# operateur qui retourne donc une valeur de type ptr...
```

Programme MPD 2: Un petit programme (pseudo-MPD) utilisant le module de dispersion d'un tableau d'entiers.

```
$ a.out 5 2
Thread  5: r =  502
Thread  1: r =  102
Thread  2: r =  202
Thread  3: r =  302
Thread  4: r =  402
```

Figure 1: Trace d'exécution pour le programme MPD 2.

```
  monitor Scatter

    op scatter( int numThread, int threadSource, ptr [*]int a ) returns int elem;

  body Scatter( int nbThreads )









    proc scatter( numThread, threadSource, a ) returns elem
    {









    }
end
```