Concurrence et collaboration

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/11/

1 Contrôle du point d'eau

On peut voir les threads comme de grands fauves (objet `Lion`) en compétition dans la jungle pour l'accès à un point d'eau (objet `Mare`). La mare est équipée d'un dispositif automatique qui enregistre chaque lion venant boire :

```
class Mare {
  private int c ;
  Mare() { c = 0 ; }
  void boire() { c++ ; }
  public String toString() { return Integer.toString(c) ; }
}
```

La classe `Lion` modélise les lions. Le comportement typique du lion est d'aller boire entre zéro et deux fois, puis d'aller se coucher (méthode `run`).

```
class Lion extends Thread {
  static Mare mare = new Mare() ;

  public void run() {
    for (int i = (int)(Math.random() * 3) ; i > 0 ; i--)
      mare.boire() ;
  }

  public static void main(String [] arg) {
    int n = 64 ;
    if (arg.length > 0) n = Integer.parseInt(arg[0]) ;
    for (int i = 0 ; i < n ; i++) {
      new Lion().run() ;
    }
    System.out.println("Bilan:␣" + mare) ;
  }
}
```

La méthode `main` procède au réveil des n lions, puis à la lecture du dispositif de la mare. Elle incarne le zoologue.

1. Un peu de Java.
   (a) Sans lire la doc, que renvoie `Math.random()` ?
   (b) Quelle est l'interface utilisateur de ce programme ? Est elle améliorable ?
   (c) Comment comprendre la construction **new** `Lion().run()` ?
2. Notre programme utlise-t-il bien plusieurs threads ?
3. On remplace **new** `Lion().run()`, par **new** `Lion().start()` et on suppose qu'il n'y a que deux lions buvant chacun une fois. Donner trois sénarios (de *scheduling*) conduisant à afficher respectivement 0, 1 et 2.
4. Corriger le problème de la question précédente. On emploiera un tableau des lions et on utilisera la méthode `join` des objets de la classe `Tread`. L'appel `t.join()` retourne normalement quand l'exécution du thread `t` est terminée.
   En fait, si `t` est toujours actif, le thread qui invoque la méthode `join` est mis en attente et `join` retourne anormalement par l'exception `InterruptedException` quand ce thread en attente est interrompu par un troisième larron (qui appelle sa méthode `interrupt`).

Proposer deux solutions, une qui échoue si le zoologue est interrompu, et une autre qui ignore les interruptions.

5. Le dispositif de comptage de la mare souffre d'un défaut. Donner un scénario conduisant à afficher 1 en présence de deux lions allant boire une fois chacun.
6. En présence du défaut, donner tous les résultats possibles dans les cas suivants :
   (a) Un lion boit deux fois.
   (b) Deux lions boivent deux fois chacun.
   (c) Un lion boit une fois et un autre deux fois.
7. Remédier au défaut.

2 J'ai rendez-vous avec vous

Soit la classe `RdV` suivante :

```
class RdV {
  boolean someone ;

  RdV () { someone = false ; }

  synchronized void sync() {
    if (someone) {
      System.out.println() ;
      someone = false ;
      notify() ;
    } else {
      someone = true ;
      try { wait() ; } catch (InterruptedException e) { System.exit(2) ; }
    }
  }
}
```

On notera que l'interuption du `wait()` se solde par un échec global de tout le programme et on en parlera plus.

1. On suppose deux threads t_0 et t_1 et un objet r de la classe `RdV` fraîchement créé. Que se passe-t-il lorsque t_0 appelle `sync()` de r, puis lorsque t_1 appelle `sync()` de r.
2. Qu'affiche le programme suivant :

```
class Two extends Thread {
  static RdV rdv = new RdV () ;
  char id ;
  Two(char id) { this.id = id ; }

  public void run() {
    for (int i = 0 ; i < 10 ; i++) {
      System.out.print(id) ; rdv.sync() ;
    }
  }

  public static void main (String [] arg) {
    new Two('A').start() ;  new Two('B').start() ;
  }
}
```

3. On dit qu'un programme termine normalement quand tous les treads qu'il a lancés terminent, sur ma machine un programe qui ne termine pas normalement ne rend pas la main, il semble bloqué. On suppose que l'objet `'A'` meurt inopinément (par exemple il déréférence bêtement **null**). Que se passe-t-il ?
4. On suppose un troisième objet **new** `Two('C')`.
   - Le programme temine-t-il normalement ?
   - De quoi sont composées les lignes de l'affichage ? On ne s'interessera qu'aux lignes complètes, terminées par un retour à la ligne et que l'affichage est synchrone, au sens où `System.out.print` ne revient pas avant que le caractère ne soit effectivement affiché.
5. On cherche à généraliser le rendez-vous à n threads. Un nouvel objet, la barrière, offre toujours une méthode `sync()`. Mais cette fois, un appel à `sync` se traduit par une attente où par un réveil collectif selon qu'il y a strictement moins de $n-1$ threads en attente ou $n-1$ (le rendez-vous est le cas particulier $n=2$). Voici un exemple d'utlisation.

```java
class All extends Thread {
  static Barrier rdv ;
  char id ;

  All(char id) { this.id = id ; }

  public void run() {
    for (int i = 0 ; i < 10 ; i++) {
      System.out.print(id) ;
      rdv.sync() ;
    }
  }

  public static void main (String [] arg) {
    int n = 26 ;
    rdv = new Barrier(n) ;
    for (int i = 0 ; i < n ; i++)
      new All((char)('A' + i)).start() ;

  }
}
```

   L'effet attendu est l'affichage de 10 lignes composées des 26 lettres de l'alphabet.
   Écrire la classe `Barrier`.
6. Le comportement de notre barrière est assez statique, dans le sens que l'on doit savoir dès le départ que l'on souhaite synchroniser n threads. On veut le rendre plus dynamique à l'aide de deux méthodes, `register()` et `checkout()` destinées respectivement à annoncer que l'on souhaite participer aux rendez-vous et que l'on y renonce (évidemment on suppose une certaine discipline de la part des threads synchronisants).
   Écrire cette nouvelle barrière à partir de la précédente.
7. Modifier la classe `All` pour utiliser la nouvelle barrière. Annoncer les affichages résultant des modifications.
8. Blinder le code pour que la mort par exception d'un thread ne déclenche pas de blocage.

Solutions des exercices

Exercice 1

1.
   (a) `Math.random()` renvoie un flottant pseudo-aléatoire (en double précision, un **double**) entre 0.0 et 1.0 (exclu). La multiplication et la conversion de type donnent finalement un entier pseudo-aléatoire entre 0 et 3 (exclu).

   (b) Le programme est normalement lancé comme `java Lion` n où n est un entier. On peut omettre l'argument n, auquel cas une valeur par défaut est substitué à n.
   L'interface n'est pas parfaite. Il faudrait :
   – Se garder des arguments non-entiers ! (Exception `NumberFormatException`).
   – Exiger 0 ou 1 argument exactement.
   – Signaler les argument absurdes (négatifs).
   Dans tous ces cas, une solution courante est de faire échouer le programme et d'afficher un message résumant l'interface utlisateur du programme.

   (c) La construction en question peut se lire comme `(new Lion()).run()`. Elle crée un objet de la classe `Lion` dont la méthode `run` est immédiatement appelée.

2. Il n'y a en fait qu'un seul thread, car `run` est invoquée directement.
3. À la différence de `run`, la méthode `start` n'exécute pas `run` elle délègue cette tâche à un nouveau thread crée pour l'occasion, et retourne tout de suite. Autrement dit l'identification des threads aux objets de la classe `Thread` a ses limites.
   Un thread est stricto-sensu une instance (de bout de) programme en cours d'exécution, et il n'est pas facile de réduire cette notion étrange à un objet.
   Bref, appelons l_0 et l_1 nos deux threads lions, nous disposons aussi d'un thread principal (celui qui exécute `main`) chargé à la fois de l'éveil des lions et de la lecture du dispositif. Voici les trois scénarios :
   (a) p s'exécute complètement avant l_0 et l_1, la valeur affichée est zéro.
   (b) p lance l_0, et donne immédiatement la main à l_0 qui s'exécute complètement, puis reprend la main, l_1 ne s'exécute qu'une fois$p$ terminé. La valeur affichée est 1.
   (c) p passe la main au lions, dès que celui-ci est lancé, puis reprend la main dès que le lion est couché. La valeur affichée est 2.
4. Évidemment on ne veut relever le dispositif qu'une fois tous les lions couchés. Voici une première solution qui échoue si le zoologue est interrompu :

```java
public static void main(String [] arg) {
    int n = 64 ;
    if (arg.length > 0) n = Integer.parseInt(arg[0]) ;
    Thread [] t = new Thread [n] ;

    for (int i = 0 ; i < n ; i++) {
      t[i] = new Lion() ;
      t[i].start() ;
    }

    try {
      for (int i = 0 ; i < n ; i++) {
        t[i].join() ;
      }
    } catch (InterruptedException e) { System.exit(2) ; }
    System.out.println("Bilan: " + mare) ;
 }
```

Pour ignorer les interruptions, il faut recomencer à attendre le coucher du lion surveillé au moment de l'interruption. La boucle finale devient :

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

On note la boucle infinie, dont on sort par **break**.

5. Le défaut peut se résumer simplement : le compteur peut compter plusieurs lions se présentant simultanément comme un seul.

   De fait l'instruction `c++` n'est pas atomique, elle peut se décomposer ainsi : d'abord, lecture de la variable partagée `c` dans une case propre à chaque thread-lion (attention c'est bien le lion qui boit, c'est son thread qui exécute le code de la méthode `boire`) ; puis incrément ; et enfin écriture de la nouvelle valeur dans la variable partagée `c`.

   On peut donc imaginer le scénario catastrophe suivant : tous les lions se présentent en même temps à la mare, et donc lisent tous la même valeur de `c`. Une fois les écritures complétées tout se passe, au vu du compteur, comme si un seul lion était passé. En terme d'ordonancement sur un seul processeur, ce scénario survient quand les lions passent systématiquement la main après la lecture de `c`.

   Toutefois, du point de vue d'un lion donné, tous ses passages sont enregistrés, puisque les lectures/écritures du compteur par un lion donné s'effectuent séquentiellement. Ainsi, la valeur finale du compteur vaut au minimun le nombre de passages à la mare d'un lion arbitraire et au maximum la somme des passages à la mare de tous les lions.

6. En fonction de la discussion précédente.
   (a) 2 passages enregistrés.
   (b) Entre 2 et 4 passages enregistrés.
   (c) Entre 1 et 3 passages enregistrés.

7. On remédie au défaut en déclarant la méthode `boire` comme **synchronized**, ainsi un seul lion peut accéder au compteur à la fois. L'effet de **synchronized** est de faire qu'au plus un thread peut se trouver dans le code de `boire` (notez que ce thread peut être *prêt* [voir le poly] c'est à dire ne s'exécutant pas). On dit aussi que le code de `boire` est en section critique.

Exercice 2

1. t_0 va se suspendre (`wait`) , puis t_1 va afficher un retour à la ligne et réveiller t_0.
2. Le programme affiche 10 lignes « `AB` » ou « `BA` » et termine normalement puisque tous les threads terminent.
3. Si `A` meurt inopinément avant son dixième rendez-vous, `B` est bloqué, et le programme ne termine pas normalement.
4. On peut grouper par paires les appels à `sync`. Soit donc $n_{\texttt{AB}}$ le nombre de rendez-vous entre `A` et `B`, etc.
   – Si le programme termine, on a $n_{\texttt{AB}} + n_{\texttt{AC}} = 10$, $n_{\texttt{AB}} + n_{\texttt{BC}} = 10$ et $n_{\texttt{BC}} + n_{\texttt{AC}} = 10$, puisque chaque objet doit effectuer 10 rendez-vous avant de terminer. Ce système a pour unique solution $n_{\texttt{AB}} = n_{\texttt{AC}} = n_{\texttt{BC}} = n_{\texttt{AC}} = 5$. Cela correspond par exemple au scénario où `A` se synchronize 5 fois avec `B`, puis 5 fois avec `C`, puis enfin `B` et `C` effectuent 5 rendez-vous. Le programme peut donc terminer. On remarque que dans ce cas 15 rendez-vous ont eu lieu.
   Il est également possible que le programme ne termine pas. Pour commencer, il est clair qu'il y a au plus un thread en attente à tout instant. Un cas de bloquage possible correspond

donc à deux threads terminés et un thread bloqué que personne ne réveillera plus. Soit par exemple, $n_{\texttt{AB}} + n_{\texttt{AC}} = 10$ et $n_{\texttt{AB}} + n_{\texttt{BC}} = 10$ (`A` et `B` ont terminé). Par soustraction on a forcément $n_{\texttt{AC}} = n_{\texttt{BC}}$. C'est à dire que `C` a réparti ses rendez-vous également entre `A` et `B`. Il peut donc y avoir entre 10 et 14 rendez-vous en cas de blocage.

– On remarque d'abord qu'une ligne ne peut pas contenir deux fois le même nom. Or, le retour à la ligne est effectué avant que les deux partenaires ne retournent du rendez-vous, essentiellement parce que l'affichage du retour à la ligne est en section critique (mais aussi parce qu'il survient avant le retour de `sync` du deuxième participant et avant le réveil du premier par `notify`). Il suffit ensuite de vérifier les quatre possibilités. (Les ordres relatifs des charactères sur une ligne sont indicatifs).

  `ABC` A et B s'affichent puis perdent la main (affichage `AB` ou `BA`), puis C s'affiche, ensuite un rendez vous intervient nécessairement et le retour à la ligne est affiché.

  `AB` A et B s'affichent puis reussissent leur rendez-vous sans que C intervienne.

  `A` La ligne précédente est `ABC`. On suppose donc qu'un rendez vous a eu lieu. On suppose en outre que A a participé au rendez vous et que le troisième thread est en attente de rendez-vous. Puis A reprend alors la main pour s'afficher et résoudre le rendez-vous avec le thread en attente.

  ligne vide Une ligne vide signifierait que les noms des deux threads réalisant le rendez-vous révélé par le retour à la ligne sont affichés sur la ligne précédente, terminée par un retour à la ligne traduisant un autre rendez-vous. C'est impossible car il n'y a que trois threads. On note que le raisonnement ne fonctionne que si il y a retour à la ligne. De fait, la dernière ligne peut être vide (si j'ose dire) si le scénario `ABC` est suivi de la terminaison des deux threads qui ont réussi leur rendez-vous.

5. Et voilà.

```java
class Barrier {
  private int nall, nwait ;

  Barrier (int nall) {
    this.nall = nall ; nwait = 0 ;
  }

  synchronized void sync() {
    nwait++ ;
    if (nwait < nall) {
      try {
          wait() ;
        } catch (InterruptedException e) {
          System.exit(2) ;
        }
    } else {
      System.out.println() ;
      nwait = 0 ;
      notifyAll() ;
    }
  }

}
```

   On remarquera surtout le remplacement d'un booléen par deux entiers et le changement de `notify` en `notifyAll`.

   Il faut un moment pour se convaincre du fonctionnement.

6. Voici la nouvelle barrière.

```java
class Barrier {
  private int nall, nwait ;

  Barrier() { nall = nwait = 0 ; }

  synchronized void register() {
    nall++ ;
  }

  synchronized void checkout() {
    nall-- ;
    if (nwait == nall && nwait != 0) {
      System.out.println() ;
      nwait = 0 ;
      notifyAll() ;
    }
  }

  synchronized void sync() {
    nwait++ ;
    if (nwait < nall) {
      try {
          wait() ;
        } catch (InterruptedException e) {
          System.exit(2) ;
        }
    } else {
      System.out.println() ;
      nwait = 0 ;
      notifyAll() ;
    }
  }

}
```

On remarque le changement de constructeur et qu'un renoncement peut entraîner un réveil.

7. Le changement porte sur d'une part sur la méthode `run` qui commence par un enregistrement et se termine par un renoncement ; et d'autre part sur la construction de la barrière qui ne dépend plus du paramètre n.

```java
class All extends Thread {
  static Barrier rdv = new Barrier() ;
  char id ;

  All(char id) { this.id = id ; }

  public void run() {
    rdv.register() ;
    for (int i = 0 ; i < 10 ; i++) {
      System.out.print(id) ; rdv.sync() ;
    }
    rdv.checkout() ;
  }

  public static void main (String [] arg) {
```

```
    int n = 26 ;
    for (int i = 0 ; i < n ; i++)
      new All((char)('A' + i)).start() ;

  }
}
```

L'affichage attendu est un certain nombre (entre 26×10 et 10) de lignes non-vides comportant des lettres distinctes. Chaque lettre apparaissant au total 10 fois nécessairement dans 10 lignes successives.

8. Le blindage demande de renoncer à la barrière en cas d'exception.

```
public void run() {
  rdv.register() ;
  try {
    for (int i = 0 ; i < 10 ; i++) {
      System.out.print(id) ; rdv.sync() ;
    }
  } catch (Throwable e) { }
  rdv.checkout() ;
}
```

On notera que les exceptions sont toutes attrapées, `Throwable` étant une classe parente de toutes les exceptions possibles. C'est un peu excessif, on pourrait n'attraper que les exceptions non signalées par le compilateur (classe `Error` ?).

On peut aussi utiliser **try** `B` **finally** `C` qui spécifie que C sera exécuté que B termine normalement ou par une exception.

```
public void run() {
  rdv.register() ;
  try {
    for (int i = 0 ; i < 10 ; i++) {
      System.out.print(id) ; rdv.sync() ;
    }
  } finally { rdv.checkout() ; }
}
```

On note une subtile différence entre les deux solutions, dans le premier cas, la méthode `run` termine normalement, dans le second elle termine sur exception. Ici, la fin de `run` mène à la mort du thread dans les deux cas et ça n'a pas d'importance, il en irait différamment dans une méthode appelée par `run`, la seconde solution est alors sans doute plus logique.