# TD/TP n$^o$4

## Threads en Java

**Exercice 1 :** Écrivez un programme Java qui crée deux threads. Chacun de ces threads doit incrémenter $n$ fois une variable globale $x$, qui vaut initialement 0. Une fois les threads finis, vous devez afficher $x$. Vous fixerez $n = 50000$. On vous demande dans cette version de ne pas utiliser de mutex. Quel résultat obtenez-vous ?

**Exercice 2 :** Écrivez un programme qui crée deux threads. Chacun de ces threads doit incrémenter $n$ fois une variable globale $x$, qui vaut initialement 0. Une fois les threads finis, vous devez afficher $x$. Vous fixerez $n = 50000$. Résolvez le problème rencontré dans l'exercice précédent en utilisant des mutex.

**Exercice 3 :** On veut repérer dans un tableau d'entiers la position du plus grand de ces entiers. On souhaite faire ce travail en parallèle, en utilisant par exemple 4 threads. Dans cet exemple, chaque thread va explorer un quart du tableau et transmet au père la position du plus grand entier trouvé dans le quart de tableau qui lui a été affecté. Le père doit alors comparer les 4 positions qui ont été retournées par les threads, déterminer quel est le plus grand des 4 entiers et afficher le résultat. Votre programme devra fonctionner avec un nombre quelconque de threads.

**Exercice 4 :** On veut écrire une procédure qui additionne deux vecteurs de réels et stocke le résultat dans un troisième vecteur passé en paramètre. On veut paralléliser l'addition en utilisant $n$ threads (vous choisirez $n = 4$).

**Exercice 5 :** On veut écrire une fonction qui calcule le produit scalaire de deux vecteurs de réels. On veut paralléliser le calcul en utilisant $n$ threads (vous choisirez $n = 4$). Chaque thread doit calculer le produit d'une partie distincte des vecteurs, et ajouter le résultat à une variable partagée par tous les threads.

**Exercice 6 :** Dans cet exercice, on veut simuler le problème que l'on a quand plusieurs threads veulent afficher des données à l'écran. On rencontre exactement ce même problème quand plusieurs processus veulent enregistrer des informations dans un fichier journal (fichier de log).

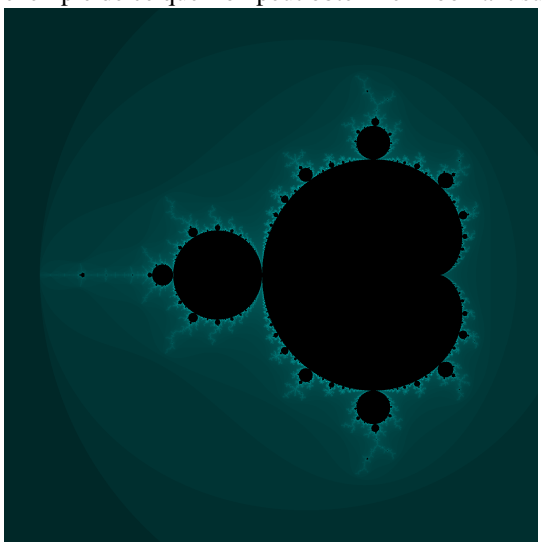
Vous devez créer trois threads, qui chacun vont afficher 20 fois une même ligne de caractères. Le premier thread devra afficher sur une ligne les lettres minuscules ('a'..'z'), le deuxième affichera les lettres majuscules ('A'..'Z') et le dernier affichera les chiffres ('0'..'9').

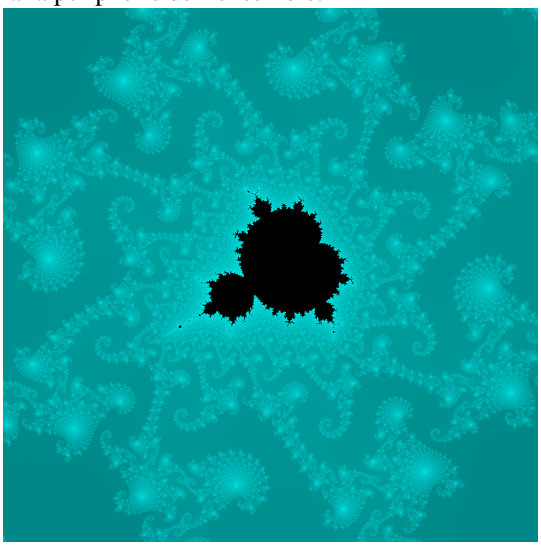
Pour bien mettre en évidence le problème qui se pose, faites une courte pause après l'affichage de chaque lettre et de chaque ligne, par exemple de 1 ms (avec `Thread.sleep(1)`).

Que se passe-t-il si on ne prend pas les précautions nécessaires ? Comment doit-on régler le problème ?

**Exercice 7 :** Dans cet exercice, on vous demande de calculer et d'enregistrer dans un fichier une image au format PPM représentant l'ensemble de Mandelbrot. Le calcul de l'image sera fait en parallèle en utilisant $n$ threads.

Ci-dessous, l'image de gauche représente l'ensemble dans sa globalité tandis que l'image de droite est un exemple de ce que l'on peut obtenir en zoomant sur des régions à la périphérie de l'ensemble.





L'ensemble de Mandelbrot est défini dans le plan complexe. Pour tout complexe $c$, on considère la série $z_{i+1} = z_i^2 + c$ avec $z_0 = 0$ et on calcule le nombre d'itérations requis pour obtenir $|z_i| > 2$. Pour certaines valeurs de $c$ (qui forment l'ensemble de Mandelbrot), cela n'arrive jamais. De ce fait, il faut imposer un nombre maximal d'itérations (par exemple 1000).

Il est d'usage de représenter l'ensemble de Mandelbrot sous la forme d'une image, qui est fractale. Chaque pixel $(x, y)$ de l'image correspond à un complexe $c = x + y.i$, et la couleur du pixel est déterminée par le nombre d'itérations requises pour obtenir $|z_i| > 2$ (en utilisant une palette de couleurs).

Le format PPM est l'un des multiples formats d'image disponible. Il est très simple à comprendre et à manipuler, mais ne permet pas de compresser les images. Ce format existe en deux versions. L'une utilise un fichier texte (format P3), l'autre utilise un fichier binaire (format P6) qui commence par un en-tête contenant du texte. Dans les deux cas, les fichiers vont contenir une matrice de pixels, chaque pixel étant défini par trois octets R,G,B donnant respectivement l'intensité de rouge (R=red), vert (G=green) et bleu (B=blue). Ces octets sont non signés, donc ont des valeurs comprises entre 0 et 255.

On précise maintenant le format texte, en simplifiant quelques détails. Vous pouvez obtenir plus de précisions avec `man ppm`.

Dans le format P3 (texte uniquement), le fichier commence par les 2 caractères "P3", suivis d'espaces (typiquement un saut de ligne), suivis de la largeur de l'image (nombre de pixels sur une ligne de l'image), d'espaces et de la hauteur de l'image. On trouve ensuite des espaces (typiquement un saut de ligne) et l'intensité maximale pour une couleur (typiquement 255 si on travaille avec un octet par couleur). Enfin, on retrouve la matrice de pixels, énumérée ligne par ligne à partir du coin supérieur gauche de l'image. Chaque pixel est représenté sous la forme de 3 nombres donnés en base 10 (R,G,B). Les caractères sont codés en ASCII.

Par exemple, pour une image de 3 pixels de large par 2 pixels de haut avec en colonne gauche du rouge, en colonne du milieu du vert et enfin du bleu en colonne droite, le fichier va contenir :

```
P3
3 2
255
255 0 0  0 255 0  0 0 255
255 0 0  0 255 0  0 0 255
```

Vous pouvez visualisez l'image que vous aurez générée avec par exemple `eog votreImage.ppm`.

**7.1:** Programmez cette génération d'images et lancez votre programme avec `time java Mandelbrot` pour observer le temps réel et le temps CPU.

Le tableau ci-dessous décrit quelques zones de l'ensemble particulièrement intéressantes.

| coordonnées complexes du centre | | largeur sur l'axe | |
|---|---|---|---|
| partie réelle | partie imaginaire | des réels | max. itérations |
| -0.7 | 0 | 3 | 1000 |
| -0.87591 | 0.20464 | 0.53184 | 1000 |
| -0.759856 | 0.125547 | 0.051579 | 1000 |
| -0.743030 | 0.126433 | 0.016110 | 1000 |
| -0.7435669 | 0.1314023 | 0.0022878 | 1000 |
| -0.74364990 | 0.13188204 | 0.00073801 | 1000 |
| -0.74364085 | 0.13182733 | 0.00012068 | 1000 |
| -0.743643135 | 0.131825963 | 0.000014628 | 1000 |
| -0.7436447860 | 0.1318252536 | 0.0000029336 | 1000 |
| -0.74364409961 | 0.13182604688 | 0.00000066208 | 10000 |
| -0.74364386269 | 0.13182590271 | 0.00000013526 | 10000 |
| -0.743643900055 | 0.131825890901 | 0.000000049304 | 10000 |
| -0.7436438885706 | 0.1318259043124 | 0.0000000041493 | 10000 |
| -0.74364388717342 | 0.13182590425182 | 0.00000000059849 | 10000 |
| -0.743643887037151 | 0.131825904205330 | 0.000000000051299 | 10000 |

**7.2:** Dans le format P6, (en-tête texte mais données binaires) le fichier commence par les 2 caractères "P6", suivis d'espaces (typiquement un saut de ligne), suivis de la largeur de l'image (nombre de pixels sur une ligne de l'image), d'espaces et de la hauteur de l'image. On trouve ensuite des espaces (typiquement un saut de ligne) et l'intensité maximale pour une couleur (typiquement 255 si on travaille avec un octet par couleur) suivi d'un seul espace (ou saut de ligne). Les caractères de l'en-tête sont codés en ASCII. Enfin, on retrouve la matrice de pixels, énumérée ligne par ligne à partir du coin supérieur gauche de l'image. Chaque pixel est représenté sous la forme de 3 octets R, G et B. Cette partie est codée en binaire.

Par exemple, pour une image de 3 pixels de large par 2 pixels de haut avec en colonne gauche du rouge, en colonne du milieu du vert et enfin du bleu en colonne droite, voici ce que va donner la commande `hexdump -C` sur le fichier :

```
00000000  50 36 0a 33 20 32 0a 32  35 35 0a FF 00 00 00 FF  |P6.3 2.255......|
00000010  00 00 00 FF FF 00 00 00  FF 00 00 00 FF           |.............|
```

Vous noterez que 0x50 est le code ASCII du caractère 'P', 0x32 correspond à '2', 0x33 correspond à '3', 0x35 correspond à '5', 0x36 correspond à '6', 0xA est le saut de ligne et enfin 0x20 est l'espace.

Pour écrire l'en-tête du fichier P6, vous commencerez par créer une chaîne de caractères $s$ contenant l'en-tête, puis utiliserez `s.getBytes(US_ASCII)` pour obtenir un tableau de bytes que vous pourrez écrire dans le fichier binaire. Vous aurez besoin d'ajouter `import static java.nio.charset.StandardCharsets.*;` pour utiliser le nom `US_ASCII`.

Un autre problème causé par Java est le suivant. On doit écrire dans le fichier des octets non signés (de 0 à 255). Or Java ne propose que des entiers signés (de -128 à +127). Pour contourner cette limitation, il faut ruser et utiliser la fonction toByte() ci-dessous. Cette fonction calcule simplement le nombre signé qui, en complément à 2, aura la représentation binaire que l'on attend. Elle découle de ce que vous avez vu en SE1.

```
/**
 * convertit un octet non signé (dans l'intervalle 0..255) en
 * entier signé (dans l'intervalle -128..127) pour que la
 * représentation binaire de cet entier signé soit égale à la
 * représentation binaire de l'octet non signé. De ce fait, quand
 * on écrit ce byte dans un fichier binaire, tout se passe comme si
 * on écrivait un octet non signé.
 */
byte toByte(int b) {
    assert(b>=0 && b<=255);
    if(b<128)
        return (byte)b;
    else
        return (byte)(256-b);
}
```

Vous pouvez examiner le contenu du fichier en hexadécimal avec la commande `hexdump -C votreImage.ppm`. Vous pouvez visualisez l'image que vous aurez générée avec par exemple `eog votreImage.ppm`.

**Exercice 8 :** On veut mettre en place une file d'entiers (structure FIFO) qui propose les méthodes put() et get() et permet de stocker au maximum $n$ entiers (on prendra $n = 20$). Différents threads appelés producteurs vont produire des nombres et les ajouter à la file (méthode put()). Différents threads appelés consommateurs vont consommer des nombres en les extrayant de la file (méthode get()). Quand la file est pleine, les producteurs doivent attendre que des places se libèrent. Quand la file est vide, les consommateurs doivent attendre qu'on y place des nombres.

Vous noterez que la file que vous mettez en place est très proche des tubes proposés par le système.

Pour tester votre programme, commencez avec une file a moitié pleine et mettez en place $p$ producteurs qui génèrent un entier à chaque seconde et $c$ consommateurs qui consomment un entier par seconde. Essayez avec $p = 2$ et $c = 2$, puis avec $p = 2$ et $c = 3$ et enfin avec $p = 3$ et $c = 2$.