TD3 : Threads en OCAML

Exercice 1 – Comptage au Musée

Le but de cet exercice est d'écrire, en OCAML/Threads, un système de comptage des entrées/sorties de visiteurs d'un musée. Le système que l'on propose sera composé de :

- Un `compteur` de type entier.
- Une fonction `entree` de thread pour compter les entrées, cette fonction prend en paramètre le nombre de visiteurs dans la file d'attente (par exemple 100).
  La signature de la fonction sera : `entree : int -> unit`
- Une fonction `sortie` de thread pour compter les sorties. La signature de cette fonction sera : `sortie : unit -> unit`

Question 1 – Variante sans synchronisation
Ecrire une implantation de ce système en OCAML/Threads, on passera le nombre de visiteurs en file d'attente sur la ligne de commande.

Question 2 – Problèmes de synchronisation
Existe-t-il des problèmes de synchronisation dans le programme ? Si oui le ou lesquels ?

Question 3 – Variante avec synchronisation
Modifier le programme pour résoudre le(s) problème(s) de synchronisation à l'aide de Mutex.

Exercice 2 – Scanneur et imprimante

Dans une bibliothèque, un système de reproduction d'ouvrage est mis en place. Ce système est composé d'un scanneur et d'une imprimante, ainsi que deux serveurs permettant de proposer des travaux de reproduction. Le programme pour contrôler ce système est donné ci dessous :

```
let printer_lock = Mutex.create ()
let scanner_lock = Mutex.create ()

let rec serveur1 nb =
        if nb > 0 then
        begin
        print_string "Serveur 1 : Job#" ;
        print_int nb ;
        print_newline();
        Mutex.lock scanner_lock ;
        Mutex.lock printer_lock ;
        print_string "Scanne\n";
        Mutex.unlock scanner_lock ;
        print_string "Imprime\n";
        Mutex.unlock printer_lock ;
        serveur1 (nb-1)
        end
        else
        print_string "Fin Serveur 1\n"

let rec serveur2 nb =
        if nb > 0 then
        begin
        print_string "Serveur 2 : Job#" ;
        print_int nb ;
        print_newline();
        Mutex.lock printer_lock ;
        Mutex.lock scanner_lock ;
        print_string "Scanne\n";
        Mutex.unlock scanner_lock ;
        print_string "Imprime\n";
        Mutex.unlock printer_lock ;
        serveur2 (nb-1)
        end
        else
        print_string "Fin Serveur 2\n"

let main () =
        let t1 = Thread.create serveur1 10
        and t2 = Thread.create serveur2 10   in
           Thread.join t1 ;
           Thread.join t2 ;
           print_string "Termine\n";
           exit 0;;

main ();;
```

Question 1 – sûreté du modèle de synchronisation
Le modèle de synchronisation proposé est-il sûr ? Si non, proposez une exécution mettant en défaut une propriété de sûreté.

Question 2 – Solutions
Proposer une solution pour corriger les problèmes éventuellement identifiés à la question précédente.

Exercice 3 – Conditions

Question 1
Donner un exemple d'affichage pour le programme suivant :

```
(* Utilisation des conditions *)
(* pour compiler :    ocamlc -thread unix.cma threads.cma cond1.ml -o cond1 *)
let cond = Condition.create ()
let cond_lock = Mutex.create ()
let task1 () =
        Thread.delay 2.0;
        (* laisse le temps aux autres de se mettre en attente *)
        Mutex.lock cond_lock;
        Condition.signal cond;
        Mutex.unlock cond_lock;
```


©2011/2012 (by UPMC/LMD/PC2R) 5 février 2017

```
        Thread.delay 2.0;
        Mutex.lock cond_lock;
        Condition.broadcast cond;
        Mutex.unlock cond_lock;
        Thread.delay 2.0;;

let task2 nb =
        Condition.wait cond cond_lock;
        print_int nb ; print_endline " => tache reveil 1";
        Mutex.unlock cond_lock;
        Condition.wait cond cond_lock;
        print_int nb ;
        print_endline " => tache reveil 2";
        Mutex.unlock cond_lock;;

let main () =
        let target1 = Thread.create task2 1
        and target2 = Thread.create task2 2
        and source = Thread.create task1 ()   in
            Thread.join source;
            print_endline "Termine";
            exit 0;;

main ();;
```

Question 2 – Question subsidiaire
Quels sont tous les affichages possibles ?

Exercice 4 – Ordonnanceur concurrent

Nous proposons de réaliser un ordonnanceur de tâches en utilisant les Threads OCAML. Complétez le programme suivant en ajoutant un mécanisme de synchronisation par condition entre le thread ordonnanceur et les threads correspondants à ces tâches.

```
(* ?????????????????????????? *)

let ordon_valeur = ref 0

let ordonnanceur nb =
  while true do
    for i = 1 to nb do
      (* ?????????????????????????? *)
      ordon_valeur:=i;
      (* ?????????????????????????? *)
    done
  done;;

let task nb =
  while true do
    (* ?????????????????????????? *)
    if(!ordon_valeur=nb)
    then
      begin
        print_string "Tache #" ; print_int nb ; print_string " activee\n";
        (* tache a realiser *)
        (* ?????????????????????????? *)
      end
    else
      (* ?????????????????????????? *)
  done;;


let main () =
  let nbtask = int_of_string Sys.argv.(1)
  in
    for i=1 to nbtask do
      let th = Thread.create task i in ();
    done ;
    let th = (Thread.create ordonnanceur nbtask)
    in
      Thread.join th;;

main ();;
```

Rappel

`synchronized` : ne fonctionne qu'avec les threads.

Dans une classe déclarée `synchronized`, ce sont toutes ses méthodes qui sont `synchronized`. Les méthodes d'instances (resp. statiques) ne peuvent s'exécuter que si l'objet (resp. la classe) auquel elles appartiennent a pu être verrouillé. `synchronized method (...)` : la methode ne pourra être executé que si elle a pu s'assurer l'exclusivité de l'utilisation de l'objet. Ce dernier est alors vérouillé. `synchronized (objet) { bloc d'instructions }` le bloc d'instructions n'est exécuté que si l'objet a pu être verrouillé.

Un thread déclaré `daemon` est un thread qui reste en mémoire pour servir d'autres threads. Il ne se termine pas de lui même. Plusieurs `daemon` peuvent s'exécuter en même temps. Tant qu'il existe un thread non `daemon` vivant, tous ces `daemons` continuent de fonctionner. S'il ne reste que les `daemons`, ces derniers seront arrêtés par la JVM.

Exercice 5 – Fil d'exécution : Runnable et Thread

Un thread vu comme suite d'instruction exécutable se définit uniquement par la méthode `run()`, de l'interface `Runnable` ; la classe `Thread` crée des objets de contrôle d'exécution ; le processus léger n'est commencé que lors de l'appel de la méthode `start()` de l'instance de contrôle.

Question 1
Définir deux classes `FilPing` et `FilPong` implémentant Runnable : la méthode `run()` affiche 20 fois `PING` (respectivement `PONG`) en rendant la main aléatoirement à en moyenne une fois sur trois.

Annexe : la méthode `nextInt()` de la classe `Random.java.util` rend un entier pseudo-aléatoire de 0 à n-1.

Question 2
Ecrire une classe `TestPingPong`, qui créé et démarre deux contrôleurs Thread exécutant `FilPing` et `FilPong`.

©2011/2012 (by UPMC/LMD/PC2R) 5 février 2017

Question 3
En TP : assurer l'alternance stricte des messages `PING` et `PONG` en utilisant un objet de classe `Bascule`(fournie) comportant un champ booléen et ses méthodes d'accès `getVal()` et de modification `setVal()`.

Exercice 6 – Le remplacement de la méthode `stop` obsolète.

[1]

On se donne une classe `Point` définie ci après :

```
class Point {
    int x, y;

    Point () {
        x = 0;
        y = 0;
    }

    void moveTo (int a_x, int a_y) {
        x = a_x;
        Thread.yield();
        y = a_y;
    }

    public String toString () {
        return "(" + x + "," +  y + ")";
    }
}
```

Question 1
Modifier alors celle ci de manière à observer le cumul de toutes les modifications faites sur `x` et sur `y` tout au long du programme. Les coordonnées ayant normalement la même valeur, on s'attend à ce que `xcumul` soit égal à `ycumul`.

Question 2
Ecrire une classe `PointThread` possédant un conctructeur `PointThread (Point a_p, int a_val)` et permettant de lancer un `thread` effectuant 20 increments de `xcumul` et `ycumul`.

Question 3
Ecrire le programme principal qui :
- crée un `Point`
- démarre 6 exemplaires de `PointThread`, avec ce `Point` et des valeurs de 1 à 6
- s'endort 10 ms
- appelle `stop` sur tous les threads sauf le premier (`val=1`)
- se rendort
- affiche les valeurs de `xcumul` et `ycumul`

Observe-t-on un blocage ? Un écart entre les deux valeurs ? Corriger éventuellement ces problèmes. On nommera la classe obtenue `PointThreadCorrect`. Relancer l'execution du programme principal en utilisant `PointThreadCorrect` observez vous un état cohérent ? Pourquoi ?

Question 4
Introduire un booléen `stopped` dans le contrôleur `PointThreadCorrect`, initialement faux, et une méthode `threadStop()` basculant se booléen à vrai.

1. voir le texte correspondant dans l'API

Question 5
Récrire la méthode `run()` de manière à consulter régulièrement le booléen `stopped` et à terminer dans un état cohérent dès qu'il est observé comme vrai. Expliquer pourquoi `xcumul` et `ycumul` sont égaux avec cette version.

Exercice 7 – Synchronisation de producteur/consommateur

Dans un schema Producteur/Consommateur, il est inutile pour le producteur de continuer à tester si la file d'attente est pleine : elle reste pleine si aucun consommateur n'intervient. Le but est donc d'attendre passivement (état bloqué) que cette condition change, à condition qu'un autre thread assure effectivement le réveil (passage à l'etat prêt) quand la valeur de cette condition change. Attention : entre le réveil et l'activation la condition peut avoir de nouveau changé de valeur : le test est toujours indispensable.

On disposera de :

```
class FifoFullException extends Exception {.......
}

class FifoEmptyException extends Exception {.......
}

class IntFifo {
IntFifo (int a_size) ...;
boolean isEmpty() ...;
boolean isFull() ...;
void add (int n) throws FifoFullException ...;
int take() throws FifoEmptyException ...;
}
```

La méthode `nextInt(n)` de la classe `Random.java.util` rend un entier pseudo-aléatoire de 0 à $n-1$.

Question 1
Ecrire une classe `Producteur` qui ajoute n fois un entier à la file d'attente `Fifo` (n est bien sûr supérieur à la taille de la file). Pour ne pas monopoliser le processeur (et permettre la consommation), il laisse la main aléatoirement de 1 à 10 pas de boucle grâce à `nextInt(10) + 1`. Par ailleurs on souhaite qu'il attende lorsque la file est pleine et assure le réveil de consommateurs(s) lorsqu'elle n'est plus vide.

Question 2
Ecrire une classe de thread Consommateur qui retire m fois un entier de la file d'attente Fifo et affiche chaque valeur reçue ; il laisse la main aléatoirement de 1 à 10 pas de boucle. On souhaite que le `Consommateur` attende lorsque la file est vide et assure le réveil de producteurs lorsqu'elle n'est plus pleine.

Question 3
Discuter les possibilités de choix entre `notify` et `notifyAll` suivant le nombre de producteurs/ consommateurs.

Question 4
En TP, écrire le programme ProdCons qui crée une file de 10 places, un producteur de 100 entiers et deux consommateurs de 50 entiers. Le programme principal attend la terminaison de tous les threads et affiche un message final avant de terminer.

Exercice 8 – Daemon (voir Annexe)

On cherche ici à simuler une file d'impression avec spooler. Le système contient un processus imprimante et des processus impression. L'imprimante doit imprimer en un seul tenant tous les textes fournis pour un processus impression. L'intérêt de l'utilisation d'un `daemon` pour l'imprimante est qu'il disparaît automatiquement lorsque tous

les processus impression sont terminés. Chaque processus impression doit attendre la fin effective de l'impression des ses pages pour quitter.

Question 1
Compléter la méthode run() de la classe `Impression` pour remplir le tableau pages avec 5 chaînes de caractères contenant "Page N du Document yyy" et envoyer ces pages au spooler de l'imprimante.

Question 2
Compléter le constructeur de la classe Imprimante pour créer un daemon.

Question 3
Compléter les méthodes `imprimer()` et `spooler(String[] s)` de la classe imprimante.

Question 4
Compléter la méthode `run()` de la classe `Imprimante` pour imprimer tant qu'il y a des pages à imprimer.

TP - Ocaml

Exercice 9 – Comptage au Musée

En reprenant l'exercice sur le musée donné en TD3, on souhaite proposer le comptage avec plusieurs entrées (>= 2) et plusieurs sorties (>= 2).

On souhaite tracer l'entrée et la sortie des visiteurs :
- Un visiteur se présente à l'entrée à l'heure xxx.
- Un visiteur entre dans le musée à l'heure xxx après avoir attendu au portique pendant xxx.
- Un visiteur se présente à la sortie à l'heure xxx.
- Un visiteur sort du musée à l'heure xxx après avoir attendu au portique pendant xxx.
- Le musée est plein.

Lorsque le musée est plein, seules les sorties sont opérationnelles, et bien sûr, les entrées sont opérationnelles dès que le musée n'est plus plein.

Question 1
Ajouter les éléments nécessaires et modifier les procédures `entree()` et `sortie()` pour ce musée.
On évitera bien sûr les attentes actives (programme) et les attentes trop longues aux portiques (visiteurs).

TP - Java

On se donne une classe `Point`, qu'on a volontairement codé pour observer des situations incohérentes :

```
class Point {
int x;
int y;

Point () {
x = 0;
y = 0;
}

void moveTo (int a_x, int a_y) {
x = a_x;
Thread.yield();
y = a_y;
}

String toString () {
return "(" + x + "," +  y + ")" ;
}
}
```

Exercice 10 – Partage de données sans exclusion mutuelle

Question 1
Ecrire une classe `PointThread` (héritant de la classe `Thread` de `java.lang`) recevant une référence `p` de type `Point` et une valeur entière `val`, dont l'exécution boucle 50 fois en redéfinissant les coordonnées de $\vec{p}$ comme `(val, val)` par `moveTo()` et affichant le point `p`.

Question 2
Ecrire le programme `TestPointThread` dont la fonction `main()` crée un point `p` et trois exemplaires de `PointThread` avec les valeurs $1, 2, 3$, puis démarre leur exécution. Compiler, exécuter, en redirigeant éventuellement les impressions dans un fichier `resP_0`.

Question 3
Observer les affichages : les coordonnées sont-elles toujours égales ?

Exercice 11 – Introduction de l'exclusion mutuelle par méthodes synchronisées dans `Point`

Copier Point.java dans `Point_0.java`

Question 1
Modifier la classe `Point` pour que chaque appel de méthode mette en oeuvre le moniteur associé à l'objet.

Question 2
Sans modifier `PointThread` et `TestPointThread`, comparer le résultat de la nouvelle exécution à celle de l'exercice 1.

Exercice 12 – Exclusion mutuelle par bloc synchronisé dans `PointThreadCorrect`, classe `Point` initiale

Copier `PointThread.java` dans `PointThreadCorrect.java`.

Question 1
Modifier la classe PointThreadCorrect pour garantir l'exclusion mutuelle avec un paramètre de classe `Point_0` (méthodes non protégées par `synchronized`).

Question 2
Copier et modifier `TestPointThreadCorrect`.

Question 3
Vérifier que l'exécution est cohérente, n'affichant que des coordonnées égales.

Exercice 13

Créer 2 threads qui écrivent n fois `ping` pour l'un et `PONG` pour l'autre en attendant un temps aléatoire entre deux affichages. La classe `java.util.Random` fournit la méthode `nextint(int max)` qui renvoie un entier entre 0 et $max - 1$. `Thread.sleep(int ms)` arrête le thread en cours d'exécution pendant `ms` millisecondes (au moins).

Question 1
Ecrire la classe de thread `PingPong`.

Question 2
Ecrire la procédure de lancement ; plusieurs `ping` - ou `PONG` peuvent-ils être écrits successivement sans occurrence de l'autre chaîne ?

Question 3
Pour garantir une alternance exacte, on introduit une classe `Sem`, protégeant une valeur booléenne avec deux méthodes `setVal()` et `getVal()` en exclusion mutuelle. Ecrire cette classe.

Question 4
L'utiliser pour garantir la stricte alternance de `ping` (vrai) et `PONG` (faux).

ANNEXE

```
public class TestImpression {
    static Imprimante    imprimante;
    static Impression [] impression;;

    public static void main (String[] args) {
        imprimante = new Imprimante();
        impression = new Impression[5];
        for (int i = 0; i < 5; i++) {
            impression[i] = new Impression("Document " + (i + 1),
                                           imprimante);
            impression[i].start();
        }
    }
}


public class Impression extends Thread {
    static Imprimante imprimante;
    String[]          pages;

    Impression (String s, Imprimante i) {
        super(s);
        imprimante = i;
    }

    public void run () {
        /* ?????????? */
    }
}


class Imprimante implements Runnable {
    String[] spooler = new String[100];
    int      index = 0;
    boolean  pagesEnAttente = false;
    Thread   daemon;


    Imprimante () {
        /* ?????????? */
    }

    /* ?????????? */

    public void run () {
        /* ?????????? */
    }
}
```


Rappel de l'API java des Threads

public class Thread extends Object implements Runnable*

Constructeurs

- `Thread()` Allocates a new Thread object.
- `Thread(Runnable target)` Allocates a new Thread object.
- `Thread(Runnable target, String name)` Allocates a new Thread object.
- `Thread(String name)` Allocates a new Thread object.
- `Thread(ThreadGroup group, Runnable target)` Allocates a new Thread object.
- `Thread(ThreadGroup group, Runnable target, String name)` Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
- `Thread(ThreadGroup group, Runnable target, String name, long stackSize)` Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size.
- `Thread(ThreadGroup group, String name)` Allocates a new Thread object.

Methodes

- `static int activeCount()` Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.
- `void checkAccess()` Determines if the currently running thread has permission to modify this thread.
- `int countStackFrames()` *Deprecated*. The definition of this call depends on suspend(), which is deprecated. Further, the results of this call were never well-defined.
- `static Thread currentThread()` Returns a reference to the currently executing thread object.
- `void destroy()` *Deprecated*. This method was originally designed to destroy this thread without any cleanup. Any monitors it held would have remained locked. However, the method was never implemented. If if were to be implemented, it would be deadlock-prone in much the manner of suspend(). If the target thread held a lock protecting a critical system resource when it was destroyed, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume *Deprecated*?.
- `static void dumpStack()` Prints a stack trace of the current thread to the standard error stream.
- `static int enumerate(Thread[] tarray)` Copies into the specified array every active thread in the current thread's thread group and its subgroups.
- `static Map<Thread,StackTraceElement[]> getAllStackTraces()` Returns a map of stack traces for all live threads.
- `ClassLoader getContextClassLoader()` Returns the context ClassLoader for this Thread.
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
- `long getId()` Returns the identifier of this Thread.
- `String getName()` Returns this thread's name.
- `int getPriority()` Returns this thread's priority.
- `StackTraceElement[] getStackTrace()` Returns an array of stack trace elements representing the stack dump of this thread.
- `Thread.State getState()` Returns the state of this thread.

— `ThreadGroup getThreadGroup()` Returns the thread group to which this thread belongs.
— `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` Returns the handler invoked when this thread abruptly terminates due to an uncaught exception.
— `static boolean holdsLock(Object obj)` Returns true if and only if the current thread holds the monitor lock on the specified object.
— `void interrupt()` Interrupts this thread.
— `static boolean interrupted()` Tests whether the current thread has been interrupted.
— `boolean isAlive()` Tests if this thread is alive.
— `boolean isDaemon()` Tests if this thread is a daemon thread.
— `boolean isInterrupted()` Tests whether this thread has been interrupted.
— `void join()` Waits for this thread to die.
— `void join(long millis)` Waits at most millis milliseconds for this thread to die.
— `void join(long millis, int nanos)` Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.
— `void resume()` *Deprecated.* This method exists solely for use with suspend(), which has been deprecated because it is deadlock-prone. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume *Deprecated* ?.
— `void run()` If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called ; otherwise, this method does nothing and returns.
— `void setContextClassLoader(ClassLoader cl)` Sets the context ClassLoader for this Thread.
— `void setDaemon(boolean on)` Marks this thread as either a daemon thread or a user thread.
— `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)` Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.
— `void setName(String name)` Changes the name of this thread to be equal to the argument name.
— `void setPriority(int newPriority)` Changes the priority of this thread.
— `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)` Set the handler invoked when this thread abruptly terminates due to an uncaught exception.
— `static void sleep(long millis)` Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
— `static void sleep(long millis, int nanos)` Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
— `void start()` Causes this thread to begin execution ; the Java Virtual Machine calls the run method of this thread.
— `void stop()` *Deprecated.* This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume *Deprecated* ?.
— `void stop(Throwable obj)` *Deprecated.* This method is inherently unsafe. See stop() for details. An additional danger of this method is that it may be used to generate exceptions that the target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For more information, see Why are Thread.stop, Thread.suspend and Thread.resume *Deprecated* ?.
— `void suspend()` *Deprecated.* This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts

to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume *Deprecated* ?.
— `String toString()` Returns a string representation of this thread, including the thread's name, priority, and thread group.
— `static void yield()` A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Methodes hérités de la classe ttclass java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`