Les documents, les calculatrices et les téléphones sont interdits.

L'évaluation des copies prendra en compte la clarté et la précision des explications produites.

Exercice 1 *(Codage d'un loquet cyclique : 7 points)* Un loquet est un objet qui comporte essentiellement un entier positif et sur lequel peuvent être appliquées deux méthodes :

- la décrémentation, notée **`dec()`**, qui décrémente la valeur courante du loquet, de manière atomique, sauf si celle-ci vaut déjà 0 ; dans ce cas, la méthode **`dec()`** est sans effet.
- l'attente, notée **`att()`**, qui bloque le thread qui l'applique jusqu'à ce que la valeur du loquet soit nulle ; dans le cas où la valeur du loquet est déjà nulle lors de l'appel à la méthode **`att()`**, celle-ci est sans effet sur le thread qui l'applique.

La valeur initiale d'un loquet est déterminée lors de sa construction.

Question 1. Écrivez *sous la forme d'un moniteur* une classe **`Loquet`** qui implémente un loquet (sans utiliser la classe **`CountDownLatch`**, ni la classe **`Phaser`**).

Question 2. Modifiez la classe **`Loquet`** obtenue de sorte que le thread qui affecte à un loquet une valeur nulle en appliquant la méthode **`dec()`**

(a) reste bloqué dans l'exécution de cette méthode tant que tous les threads préalablement en attente sur le loquet n'ont pas tous redémarré ;

(b) puis, avant de quitter la méthode **`dec()`**, repositionne le loquet à sa valeur initiale.

En outre, pendant la phase de redémarrage des threads préalablement en attente, aucun autre thread ne peut accéder au loquet, ni par la méthode **`att()`**, ni par la méthode **`dec()`**.

```
while ( !  arrêtDuServeur ) {
  Requête r = accepterUneRequête();
  r.traiterUneRequête();
}
```

FIGURE 30 – Code simplifié d'un serveur

Exercice 2 *(Réservoir de threads : 3 points)* Dans un système client-serveur, le rôle du serveur est d'accepter chaque requête soumise et d'y répondre. Le code simplifié d'un serveur est indiqué sur la figure 30 : chaque appel à la méthode **`accepterUneRequête()`** renvoie une nouvelle requête acceptée (et modélisée par un objet de la classe **`Requête`**) à laquelle il est répondu par l'application de la méthode **`traiterUneRequête()`** sur cette requête. Écrire un code alternatif à celui de la figure 30 qui utilise un réservoir de 4 threads afin de permettre le traitement de plusieurs requêtes en parallèle.

```
class MonVerrou {
  private Thread possesseur = null;
  public synchronized void verrouiller() throws InterruptedException {
    while ( possesseur != null ) wait();
    possesseur = Thread.currentThread();
  }
  public synchronized void déverrouiller() {
    if ( possesseur == Thread.currentThread() ) possesseur = null;
    notifyAll();
  }
}
```

FIGURE 31 – Code artisanal d'un verrou (non-réentrant)

Exercice 3 *(Programmation sans verrou : 5 points)* La figure 31 décrit la classe d'un verrou non-réentrant, implémentée sous la forme d'un moniteur. Écrire le code d'une classe équivalente *sans utiliser de verrou*, à l'aide d'objets atomiques. L'emploi du mot-clef **`synchronized`** ou de la classe **`ReentrantLock`** est donc proscrit.

Exercice 4 *(États d'un thread : 2 points)* Décrire l'exécution du programme de la figure 32 à l'aide d'un diagramme de séquence. Indiquez ensuite, dans l'ordre, la sortie écran obtenue.

```
class Test {
  static Object objet = new Object();
  static Thread Cobaye, Observateur;
  public static void affiche(){
    System.out.println("Etat du thread Cobaye: " + Cobaye.getState());
  }
  public static void main(String[] args) throws InterruptedException {
    Cobaye = new Thread (
               new Runnable(){
                 public void run() {
                     try {
                       synchronized(objet){ objet.wait(2000); }
                     } catch(InterruptedException ignoree){}
                   }
                 });
    Observateur = new Thread(
                    new Runnable(){
                      public void run(){
                        try {
                          Thread.sleep(1000) ;
                          affiche();
                          synchronized(objet){
                            Thread.sleep(2000) ;
                            affiche();
                          }
                        } catch(InterruptedException ignoree){}
                      }
                    });
    affiche();
    Cobaye.start();
    Observateur.start();
    Cobaye.join();
    affiche();
    Observateur.join();
  }
}
```

FIGURE 32 – États d'un thread

Exercice 5 *(Algorithmique parallèle : 3 points)* Le problème du sous-tableau maximum consiste à calculer, au sein du tableau `a`, une suite d'éléments contigus dont la somme est maximale. Formellement, il s'agit de déterminer la valeur maximale de la somme `a[i]+a[i+1]+...+a[j-1]` lorsque les indices `i` et `j` varient en respectant la contrainte $0 \leqslant$ `i` $\leqslant$ `j` $\leqslant n$, où n désigne la longueur du tableau. Cette somme est nulle lorsque `i` est égal à `j`. L'algorithme de Kadane résoud ce problème en temps linéaire à l'aide de la programmation dynamique. Une implémentation de cet algorithme est donnée sur la figure 33.

Question 1. Proposez une approche susceptible d'améliorer le temps de calcul en employant deux threads.

Question 2. Écrire en détails un code Java qui implémente votre proposition.

```
int sousSequenceMaximale(int a[], int taille) {
  int segmentMaximal = 0;
  int suffixeMaximal = 0;
  for (int i = 0; i < taille; i++) {
    suffixeMaximal = Math.max(0, suffixeMaximal + a[i]);
    segmentMaximal = Math.max(segmentMaximal, suffixeMaximal);
  }
  return segmentMaximal;
}
```

FIGURE 33 – Codage en Java de l'algorithme de Kadane