# Programmation multi-cœurs — TD 2
# Programmation bloquante

**Exercice 1.** *La pile bloquante*. Une pile bloquante bornée, dont la capacité est passée en paramètre au constructeur, possède deux opérations :

— `void push(T)` attend qu'il y ait de la place dans la pile puis ajoute un élément en tête de pile ;

— `T pop()` attend que la pile ne soit pas vide puis supprime et retourne l'élement en tête de pile.

Implémentez une pile bloquante bornée en Java.

**Exercice 2.** *Verrouillage imbriqué de moniteur*. Quel est le problème du code suivant ? (Une version complète est disponible dans le fichier `examples.NestedLockout.java` sur Madoc.)

```
class FairLock {
  List<Thread> waitingThreads = new ArrayList<Thread>();

  public synchronized void lock() throws InterruptedException {
    synchronized(waitingThreads){
      waitingThreads.add(Thread.currentThread());
      while(waitingThreads.get(0) != Thread.currentThread())
      waitingThreads.wait();
    }
  }

  public synchronized void unlock(){
    synchronized(waitingThreads){
      waitingThreads.remove(0);
      waitingThreads.notify();
    }
  }
}
```

**Exercice 3.** *Bande de sauvages*. Une tribu de sauvages mange son dîner dans un pot commun qui peut contenir $M$ parts de potage. Quand un sauvage veut manger, il se sert au pot, sauf s'il est vide. Quand le pot est vide, les sauvages réveillent le cuisinier et attendent qu'il remplisse le pot.

Programmer le comportement des sauvages et du cuisinier. Les contraintes sont les suivantes :

— Les sauvages ne peuvent pas se servir dans un pot vide ;

— Le cuisinier ne doit remplir le pot que quand il est vide ;

— Discutez les propriétés de vivacité obtenues.

**Exercice 4.** *Arbre binaire de recherche*. On cherche à implémenter un arbre binaire de recherche (ABR) en se basant sur le code suivant. Identifier les problèmes de concurrence qui peuvent se poser, puis proposez un moyen de les résoudre.

```
public class ABR {
  private Node root = new Node();

  public boolean contains(int v) {
    return root.get(v).isSet;
  }

  public void insert(int v) {
    Node n = root.get(v);
    if(!n.isSet) n.setValue(v);
  }

  public void delete(int v) {
    Node n = root.get(v);
    if(!n.isSet) return;
    if(!n.smaller.isSet) n.copyNode(n.greater);
    else {
      Node predecessor = n.smaller;
      while(predecessor.greater.isSet)
      predecessor = predecessor.greater;
      n.value = predecessor.value;
      predecessor.copyNode(predecessor.smaller);
    }
  }
}

class Node {
  Node smaller = null;
  boolean isSet = false;
  int value;
  Node greater = null;

  void copyNode(Node n) {
    isSet = n.isSet; value = n.value; smaller = n.smaller; greater = n.greater;
  }
  void setValue(int v) {
    isSet = true; value = v; smaller = new Node(); greater = new Node();
  }

  Node get(int v) {
    if(!isSet || v == value) return this;
    if (v < value) return smaller.get(v);
    return greater.get(v);
  }
}
```