

Mémoire partagée et sémaphores en java

Frédéric Guinand

Département informatique / IUT du Havre

Avoir plusieurs threads pose des problèmes et des questions

Synchronisation

- comment les différents threads peuvent se synchroniser ?
- comment plusieurs threads peuvent partager des données ?

les données risquent de perdre leur intégrité si deux threads les manipuler en "même temps"

- Si chaque thread manipule des données qui lui sont propres alors pas de problème !
- Si plusieurs threads manipulent des données communes alors :
 - Problème :
 - Espace de travail commun, pas de mémoire privée pour chaque thread
 - Accès simultané à une même ressource
 - Les données risquent leur intégrité si deux threads se retrouvent à les manipuler en même temps
 - Exemple :
 - On considère une bonbonnière et un ensemble d'enfants qui peuvent avoir accès à la bonbonnière sans restriction

Exemple

- on considère une seule bonbonnière (objet partagé)
- les enfants sont des threads
- un enfant ne prend qu'un bonbon à la fois dans la bonbonnière
- à chaque demande de bonbon, la bonbonnière :
 - ❶ délivre un bonbon numéroté à l'enfant
 - ❷ décrémente le nombre de bonbons qui restent

```
public class Principale {  
    instancie un objet Bonbonniere en  
        passant le nombre initial de bonbons en paramètre  
        au constructeur de Bonbonniere  
    instancie k objets Enfant (chaque Enfant est un thread) en  
        passant à leur constructeur la référence à Bonbonniere  
        et un identifiant entier  
}
```

Exemple

```
class Bonbonniere {  
    int nbBonbons ;  
    public Bonbonniere(...) { ... }  
    public int attrapeBonbon() {  
        int bonbon = 0 ;  
        if ( nbBonbons > 0 ) {  
            bonbon = nbBonbons ; // instruction A  
            nbBonbons -- ; // instruction B  
        }  
        return bonbon ;  
    }  
}  
class Enfant extends Thread {  
    public Enfant(...) { ... }  
    public void run() {  
        int bonbon = bonbonniere.attrapeBonbon() ;  
        while(bonbon != 0) {  
            S.out.P("enfant :" + id + " bonbon : " + bonbon) ;  
            bonbon = bonbonniere.attrapeBonbon() ;  
            nb++ ;  
        }  
        S.out.P("nb bonbons mangés : " + nb) ;  
    }  
}
```

- Exécutez ce code.
- Que constatez-vous ?
- Modification : vous ajoutez maintenant entre les instructions A et B du code de la Bonbonnière un bout de code qui simule un traitement un peu long. Que constatez-vous ?

Constatations

- ① les messages sont affichés dans un ordre quelconque
- ② tous les enfants ne mangent pas le même nombre de bonbons
- ③ certains bonbons sont récupérés plusieurs fois, d'autres n'apparaissent jamais
- ④ le nombre de bonbons consommés est différent du nombre initial de bonbons dans la bonbonnière !!

Diagnostic

- les points 1 et 2 relèvent de l'indéterminisme et correspondent à un comportement normal d'un système réparti/distribué, par contre...
- les points 3 et 4 sont les symptômes d'un problème, dans la logique de l'application, chaque bonbon doit être consommé une fois et une seule

Problèmes

- 1 certains bonbons sont récupérés plusieurs fois, d'autres n'apparaissent jamais
- 2 le nombre de bonbons consommés est différent du nombre initial de bonbons dans la bonbonnière !!

Solution

- garantir l'accès exclusif à un objet pendant l'exécution de la méthode
⇒ synchronized permet de gérer les concurrences d'accès

```
class Bonbonniere {  
    int nbBonbons ;  
    public Bonbonniere(...) { ... }  
    public synchronized int attrapeBonbon() {  
        int bonbon = 0 ;  
        if ( nbBonbons > 0 ) {  
            bonbon = nbBonbons ;  
            ... /* long calcul */  
            nbBonbons -- ;  
        }  
        return bonbon ;  
    }  
}
```

Synchronisation (1/2)

Lors de l'appel à une méthode `synchronized`

- l'objet est-il accessible (non-verrouillé) ?
 - Si oui, l'objet est verrouillé et le thread exécute la méthode
 - Si non, le thread doit attendre la levée du verrou (c'est-à-dire que l'exécution de la méthode (verrouillée) par un autre thread soit terminée)
- On appelle **section critique** :
 - Une méthode : déclaration précédée de `synchronized`
 - Une instruction (ou un bloc) : précédée de `synchronized`
 - Un objet : le déclarer `synchronized`

Synchronisation (2/2)

- Pour gérer la concurrence d'accès à une méthode :
 - Si un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet
 - En revanche, il peut exécuter cette méthode pour un autre objet

```
public synchronized void maMethode() { ... }
```

- Pour Contrôler l'accès à un objet :

```
public void maMethode() { ...  
    synchronized(objet) {  
        objet.methode();  
    }  
}
```

- l'accès à l'objet passé en paramètre de `synchronized(Object)` est réservé à un unique thread

Exercice

- testez ce code, allongez les suites de 0 et de 1 jusqu'à observer des coupures puis
- modifiez-le pour que l'affichage des chaînes de 0 ou de 1 ne soit pas interrompu.

```
class Affichage {  
    public void afficher(String t) {  
        for (int i=0 ; i<t.length() ; i++)  
            System.out.print(t.charAt(i)) ;  
        System.out.println() ;  
    }  
}  
public class TPrint extends Thread {  
    static Affichage mAff = new Affichage() ;  
    String txt ;  
    public TPrint(String t) {txt = t ;}  
    public void run() {  
        for (int j=0 ; j<20 ; j++)  
            mAff.afficher(txt) ;  
    }  
}  
public static void main(String args[]) {  
    new TPrint("0000000000000000000000000000").start() ;  
    new TPrint("1111111111111111111111111111").start() ;  
}
```

Limites de *synchronized*

- l'utilisation de *synchronized* ne garantie pas l'ordre dans lequel les threads accèdent à la ressource partagée
- c'est l'ordonnanceur qui décide quel thread sera la prochain à exécuter du code
 - ⇒ si l'ordre d'accès est important il faut un mécanisme supplémentaire
- par exemple si deux threads doivent exécuter un code à tour de rôle, *synchronized* ne permet pas à lui tout seul de régler le problème.

Exemple illustrant les limites de *synchronized*

- on considère deux threads qui doivent effectuer une action à tour de rôle, comme les échanges au tennis ou au ping-pong
- ils ont en partage la balle
- essayez de coder cet exemple avec la Balle qui est l'objet partagé, et deux Joueurs (deux threads)
- comment gérez-vous l'alternance des actions des deux threads sur la Balle ?
- indication : la balle doit garder en mémoire l'identifiant du dernier **thread** qui l'a accédée

Exemple

```
class Joueur extends Thread {  
  
    int id;  
    int score = 0;  
    Random alea;  
    Balle balle;  
  
    public Joueur(int id, Balle b) {  
        this.id = id; balle = b;  
    }  
  
    public void run() {  
        while(balle.tape(id)) {  
            score++;  
        }  
        System.out.println("Joueur "+id+" a terminé après "+score+" échanges");  
    }  
}
```

Exemple

```
class Balle {  
  
    boolean encours = true;  
    Random alea;  
    int echange = 0;  
  
    public Balle() { alea = new Random(System.currentTimeMillis()); }  
  
    public synchronized boolean tape(int idJoueur) {  
  
        if(encours) {  
            if(alea.nextInt(100) > 20) { // echange continu  
                System.out.println("Joueur "+idJoueur+" a réussi à taper la balle");  
                echange++;  
            } else {  
                encours = false;  
                System.out.println("Joueur "+idJoueur+" a raté la balle");  
                System.out.println("longueur de l'échange "+echange);  
            }  
        }  
        return encours;  
    }  
}
```

Exemple (traces d'exécution)

```
public class PingPong {  
  
    Joueur j1, j0;  
  
    public static void main(String[] args) {  
        new PingPong();  
    }  
  
    public PingPong() {  
        Balle b = new Balle();  
        j0 = new Joueur(0,b);  
        j1 = new Joueur(1,b);  
        j0.start();  
        j1.start();  
    }  
}
```

Joueur 0 a réussi à taper la balle
Joueur 0 a raté la balle
longueur de l'échange 4
Joueur 0 a terminé après 4 échanges
Joueur 1 a terminé après 0 échanges

pas vraiment des échanges...

Ordre d'exécution des threads (via un objet partagé)

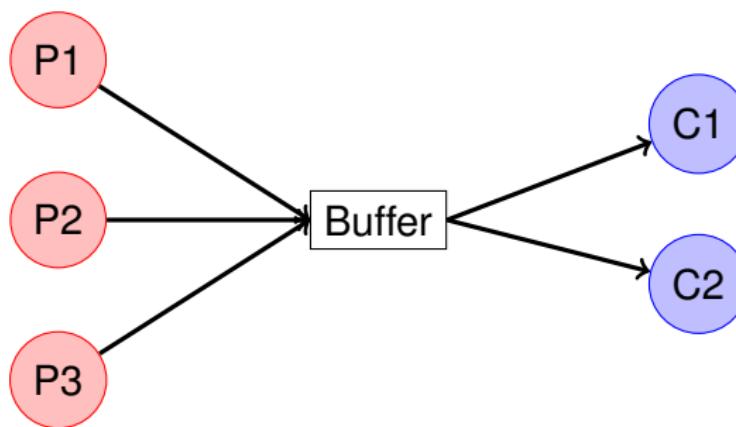
- pour résoudre ce problème, une solution consiste à mettre en attente le dernier thread qui a eu accès à la ressource partagée, ce qui peut être réalisé par la primitive `wait()`
→ `try { wait(); } catch(InterruptedException ie) {}`
- il faut également un mécanisme de réveil du thread en attente lorsque l'accès a été libéré, c'est réalisé par la primitive `notify()` (le premier thread en attente) ou `notifyAll()` (tous les threads en attente sont prévenus)

- `wait()` :
 - Doit être appelée depuis l'intérieur d'une méthode ou d'un bloc `synchronized`
 - Attend l'arrivée d'une condition sur l'objet sur lequel elle s'applique
 - Le thread en cours d'exécution est bloqué
 - Pendant que le thread attend, le verrou est relâché
 - Il existe aussi `wait(long temps)` qui termine l'attente après temps millisecondes, si cela n'est pas fait auparavant
- `notify()` / `notifyAll()` :
 - `notify()` : libère le thread dans l'état `wait`
 - Si plusieurs threads sont dans l'attente imposée, `notify()` débloque celui qui attend le plus longtemps
 - `notifyAll()`, fait la même chose mais pour tous les threads en attente

```
class Balle {  
  
    boolean encours = true;  
    int joueurencours = 0;  
    Random alea;  
    int echange = 0;  
  
    public Balle() {  
        alea = new Random(System.currentTimeMillis());  
        if(!alea.nextBoolean()) joueurencours = 1;  
        System.out.println("engagement pour le joueur "+joueurencours);  
    }  
  
    public synchronized boolean tape(int idJoueur) {  
  
        while(idJoueur != joueurencours) { try { wait(); } catch(InterruptedException ie) {} }  
        if(encours) {  
            if(alea.nextInt(100) > 20) { // echange continu  
                System.out.println("Joueur "+joueurencours+" a réussi à taper la balle");  
                echange++;  
            } else {  
                encours = false;  
                System.out.println("Joueur "+joueurencours+" a raté la balle");  
                System.out.println("longueur de l'échange "+echange);  
            }  
        }  
        joueurencours = (joueurencours+1)%2;  
        notify();  
        return encours;  
    }  
}
```

Producteurs / Consommateurs

- ceci nous amène à un problème très fréquent en informatique,
- problème que l'on retrouve des couches logicielles basses (système d'exploitation) jusqu'au niveau applicatif :
→ le problème **Producteur/Consommateur**, aussi connu sous le nom **Lecteur/Rédacteur**.



Principe

- un ensemble de threads accèdent à un espace partagé pour y déposer des items
 - les **producteurs**
- d'autres threads y accèdent pour retirer ces items afin de les traiter
 - les **consommateurs**
- un problème se pose lorsque :
 - ➊ des consommateurs tentent de retirer des items de la zone de stockage et que celle-ci est vide
 - ➋ des producteurs tentent d'ajouter des items à une zone ayant atteint sa capacité maximale de stockage

Exercice

- Les producteurs/consommateurs partagent un buffer de capacité égale à $N > 0$.
 - Les producteurs et les consommateurs doivent se synchroniser sur le buffer.
 - Un producteur doit attendre que le buffer ne soit pas plein pour ajouter un item dans le buffer.
 - Un consommateur attend que le buffer ne soit pas vide pour consommer un item.
-
- En vous inspirant de l'exemple du ping-pong et en utilisant le mot clef *synchronized* ainsi que les méthodes `wait()` et `notify()` (et éventuellement `notifyAll()`) proposez une solution
 - la solution sera composée de quatre classes : Buffer, Producteur, Consommateur et Lanceur (qui contient le main).
 - les classes Producteur et Consommateur étendent la classe Thread

Tous les problèmes d'accès à des ressources partagées sont-ils résolus ? **NON**

Exemple

- considérons deux équipes de deux personnes : $\{A_1, A_2\}$ et $\{B_1, B_2\}$,
- pour se contacter, chaque équipe doit disposer de deux téléphones, or il n'y a que deux téléphones
- hypothèse, lorsqu'une personne a accès à un téléphone, elle ne le lâche qu'à l'issue de la communication avec son partenaire,
- que se passe-t-il si A_1 et B_2 ont accès chacun à un téléphone ?

on se trouve dans une situation **d'interblocage**

- Producteur.java ∪ Consommateur.java → ProdConso.java
- Buffer.java reste inchangé
- Dans la méthode run(), le fait de produire ou de consommer est choisi aléatoirement à chaque tour de boucle
- au moment de l'instantiation des ProdConsos, 3 paramètres sont envoyés : un identifiant, une référence au buffer et le nombre d'opérations à effectuer (production et/ou consommation) par l'objet
- lancez le programme avec un buffer initialement à 100 (avec limite MAX à 500) et 5 ProdConsos : que se passe-t-il ?
- Tout se passe bien ? Alors limiter le nombre de ProdConsos à 3 et lancez le programme avec un buffer initialement à 0.

```
public class ProdConso extends Thread {  
  
    int id, nbprodcons;  
    Random alea;  
    Buffer buffer;  
  
    public ProdConso(int id, Buffer buffer, int nbprodcons) {  
        this.id = id; this.nbprodcons = nbprodcons;  
        this.buffer = buffer;  
        alea = new Random(System.currentTimeMillis());  
    }  
  
    public void run() {  
        int currentnb = 0;  
        while(currentnb < nbprodcons) {  
            if(alea.nextBoolean()) { // producteur  
                System.out.println("prodcons:"+id+" va produire");  
                buffer.ajouter();  
                System.out.println("prodcons:"+id+" a fini de produire: nbprodcons="+currentnb);  
                currentnb++;  
            } else { // consommateur  
                System.out.println("prodcons:"+id+" va consommer");  
                buffer.retirer();  
                System.out.println("prodcons:"+id+" a fini de consommer: nbprodcons="+currentnb);  
                currentnb++;  
            }  
        }  
        System.out.println("prodcons:"+id+" a TERMINE");  
    }  
}
```

```
public class Lanceur {  
  
    Buffer buf;  
    ProdConso[] prodconsos;  
  
    public static void main(String[] args) {  
        new Lanceur(3);  
    }  
  
    public Lanceur(int nbProdConso) {  
        buf = new Buffer(10);  
        prodconsos = new ProdConso[nbProdConso];  
        for(int p=0;p<nbProdConso;p++) {  
            ProdConso pc = new ProdConso(p,buf,5);  
            prodconsos[p] = pc;  
            pc.start();  
        }  
    }  
}
```

Observation

- Testez le programme plusieurs fois.
- Que constatez-vous ?
- Comment l'expliquez-vous ?

3 instances de ProdConso buffer initialement à 10

```
prodcons :1 va produire
prodcons :2 va produire
prodcons :0 va produire
----- contenu buffer 11
prodcons :1 a fini de produire : nbprodcons=0
----- contenu buffer 12
prodcons :0 a fini de produire : nbprodcons=0
prodcons :0 va consommer
prodcons :1 va consommer
----- contenu buffer 13
prodcons :2 a fini de produire : nbprodcons=0
prodcons :2 va consommer
----- contenu buffer 12
prodcons :1 a fini de consommer : nbprodcons=1
prodcons :1 a TERMINE
----- contenu buffer 11
prodcons :0 a fini de consommer : nbprodcons=1
prodcons :0 a TERMINE
----- contenu buffer 10
prodcons :2 a fini de consommer : nbprodcons=1
prodcons :2 a TERMINE
```

- 5 instances de ProdConso buffer initialement à 0

```
prodcons :0 va consommer
prodcons :3 va consommer
prodcons :2 va consommer
prodcons :1 va consommer
prodcons :4 va consommer
```

→ blocage

Autres problèmes

Stationnement

- supposons que nous disposons d'un parking de n places
- comment gérer ces n places ?
- le problème est que cette fois notre ressource peut-être accédée **simultanément** par n voitures, ce que ne permet pas *synchronized*



Guichets de banque

- les banques sont généralement pourvues d'une salle d'attente avec une certaine capacité
 ⇒ cette salle peut être accédée par plusieurs personnes à la fois
- lorsqu'un employé de banque est disponible il ne peut s'occuper que d'un client à la fois et ce client est déjà présent dans la salle d'attente
- comment peut-on reproduire cette situation ?



Sémaphores

Constats

- dans certaines situations, l'accès à des ressources partagées peut mener à des **interblocages**
- dans certains cas, l'accès à une ressource partagée peut être autorisé à **plus d'un thread à la fois**

Solution

- dans certains cas il s'agit d'un **problème de conception** (interblocage)
- les **sémaphores** peuvent apporter une réponse aux problèmes précédents

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

Sémaphores

- les **sémaphores** sont un mécanisme introduit par **Edsger W. Dijkstra**
- un sémaphore est un mécanisme qui permet de synchroniser les threads,
- début des années 60 !!



Edsger Wybe Dijkstra

Quelques contributions :

- 1 un algorithme pour le calcul du plus court chemin dans un graphe (1959)
- 2 l'élimination de l'instruction **goto** au profit de structure de contrôle (*if...then...else, while, repeat...until*) (1968)
- 3 systèmes auto-stabilisateurs (1974)

Principe

- un sémaphore S est associé à une valeur entière *semval* positive ou nulle, en java la terminologie choisie est celle de *permits*,
- lorsque cette valeur est positive, le thread peut accéder à la zone protégée par le sémaphore,
- lorsque cette valeur est nulle, un tel accès n'est pas possible, le thread est bloqué en attendant que la valeur soit strictement positive,
- deux opérations sont donc définies **P(S)** (*Passering* : passer) et **V(S)** (*Vrijgave* : relâcher), la première pour accéder à la ressource et la seconde pour la libérer
→ les **opérations atomiques de Dijkstra**

Primitive P(S) – logique interne du sémaphore

Si $semval = 0$ alors

 le thread est en attente

Sinon

$semval \leftarrow semval - 1$

 suite du traitement par le thread

finSi

Primitive V(S) – logique interne du sémaphore

$semval \leftarrow semval + 1$

 débloquer le premier thread en attente

Création et opérations élémentaires

- `new Semaphore(nbPermits, true)` → garantie FIFO (premier thread demandeur = premier thread servi)
- pour accéder à la section critique, le thread doit invoquer la méthode `acquire()` sur le sémaphore
- pour relâcher le sémaphore le thread doit invoquer la méthode `release()`
- un thread peut éventuellement demander plusieurs *permits* en invoquant `acquire(nb permis)`
- pour connaître le nombre de permis disponibles invoquez la méthode : `availablePermits()`
- pour avoir une estimation du nombre de threads en attente on peut invoquer la méthode : `getQueueLength()`

Analogie Commerce ↔ Section Critique

- Lors de la pandémie (Covid-19), durant une certaine période, les **commerces** doivent **réguler le nombre de personnes qui peuvent simultanément être présents** dans leur magasin, on appelle cette limite la **jauge**
 - pour mettre en place ce système, les **commerces** placent un **gardien** à l'entrée du magasin avec des **jetons**, dont le nombre est égal à la jauge
 - à chaque nouveau **client** entrant le gardien donne un jeton et le récupère de chaque personne qui sort
 - lorsque le gardien n'a plus de jeton en main, aucun nouveau client ne peut entrer, il doit "faire la queue" et attendre que le gardien récupère des jetons
-
- dans notre cas, le **commerce** correspond à la **section critique**, à l'**objet partagé** par plusieurs threads
 - le **gardien** joue le rôle du **sémaphore**
 - la **jauge** correspond au nombre de **permits**
 - chaque **client** qui arrive **demande un jeton au gardien**, de manière équivalente, le **thread** invoque la méthode **acquire()**
 - et lorsque le **client** quitte le commerce il **rend le jeton au gardien**, dans notre cas, le **thread** invoque la méthode **release**
 - Notez que c'est **le commerce** (**la section critique/l'objet partagé**) qui met en oeuvre le **gardien** (**le sémaphore**)



Un sémaphore particulier : le **mutex**

- un **sémaphore** initialisé à 1 est un **sémaphore binaire**, il permet de limiter l'accès à une zone de code, ou à une ressource, à un thread à la fois,
- on parle d'**exclusion mutuelle**, ce sémaphore est aussi appelé **mutex**

Exemple

- exemple construit avec un objet possèdant une section critique accessible par un nombre limité de threads (3 dans notre exemple) et 5 threads.

```
package iut.semaphore.exemple;  
  
import java.util.concurrent.Semaphore;  
  
public class ExempleSemaphore {  
  
    public static void main(String[] args) {  
        Critique critique = new Critique(3);  
        for(int i=0;i<5;i++)  
            new UnThread(critique,4).start();  
    }  
}
```

```
// ===== Les Threads =====  
class UnThread extends Thread {  
  
    Critique c;  
    int nbpassages;  
    public UnThread(Critique c, int nbpassages) {  
        this.c = c;  
        this.nbpassages = nbpassages;  
    }  
  
    public void run() {  
        int nb = 0;  
        while(nb < nbpassages) {  
            c.sectionCritique();  
            nb++;  
        }  
    }  
}
```

Exemple

```
// ===== Objet avec Section Critique =====
class Critique {

    Semaphore cerbere;
    public Critique(int nbaccesmax) {
        cerbere = new Semaphore(nbaccesmax, true);
        System.out.println("====> nb de permis : "+cerbere.availablePermits());
    }

    public void sectionCritique() {
        try {
            cerbere.acquire();
            System.out.println(Thread.currentThread().getName()
                +" est entré, il reste "+cerbere.availablePermits()+" permis");
            System.out.println("\t"+Thread.currentThread().getName()
                +" fait qqch dans la section critique");
            Thread.sleep((int)(Math.random()*100));
        } catch(Exception e) {}
        System.out.println(Thread.currentThread().getName()+" va sortir");
        cerbere.release();
    }
}
```

```
====> nb de permis : 3
Thread-0 est entré, il reste 2 permis
Thread-0 fait qqch dans la section critique
Thread-2 est entré, il reste 0 permis
Thread-2 fait qqch dans la section critique
Thread-1 est entré, il reste 1 permis
Thread-1 fait qqch dans la section critique
Thread-2 va sortir
Thread-3 est entré, il reste 0 permis
Thread-3 fait qqch dans la section critique
Thread-0 va sortir
Thread-4 est entré, il reste 0 permis
Thread-4 fait qqch dans la section critique
Thread-1 va sortir
Thread-2 est entré, il reste 0 permis
Thread-2 fait qqch dans la section critique
Thread-4 va sortir
Thread-0 est entré, il reste 0 permis
Thread-0 fait qqch dans la section critique
Thread-3 va sortir
Thread-1 est entré, il reste 0 permis
Thread-1 fait qqch dans la section critique
Thread-0 va sortir
Thread-4 est entré, il reste 0 permis
Thread-4 fait qqch dans la section critique
Thread-2 va sortir
Thread-3 est entré, il reste 0 permis
Thread-3 fait qqch dans la section critique
Thread-1 va sortir
Thread-0 est entré, il reste 0 permis
Thread-0 fait qqch dans la section critique
Thread-3 va sortir
Thread-4 va sortir
Thread-2 est entré, il reste 0 permis
```

Exercice

- revenons à notre problème du parking
- on considère un parking (objet partagée) disposant de n places
- on considère un ensemble de $k > n$ voitures (threads)
- comment gérer l'accès des véhicules au parking ?

Contraintes

- une voiture qui accède au parking reste dans le parking pendant un temps aléatoire,
- proposez une solution pour ce problème en utilisant un seul sémaphore et sans utiliser le mot clef *synchronized*,
- affichez le nombre de *permits* et le nombre de threads en attente.

les interblocages

- dans certaines situations, l'accès à des ressources partagées peut mener à des **interblocages**
- c'est le cas par exemple lorsque deux threads T_1 et T_2 ont besoin, chacun, d'accéder à deux ressources distinctes A et B chacune ayant un accès restreint par un sémaphore, il peut alors se passer le phénomène suivant :
 - 1 T_1 obtient le permis de la ressource A
 - 2 T_2 obtient le permis de la ressource B
 - 3 T_1 est en attente de la ressource B
 - 4 T_2 est en attente de la ressource A
- chaque thread est en attente de la ressource détenue par l'autre thread \Rightarrow **interblocage**
- l'exemple suivant présente un code qui illustre cette situation

Exemple d'interblocage

- exemple construit avec deux objets possèdent une section critique dont la gestion est assurée par un mutex et 2 threads.

```
import java.util.Random;
import java.util.concurrent.Semaphore;

public class InterblocageSemaphore {

    public InterblocageSemaphore(int nbressources, int nbthreads) {
        DeuxRaquettes dr = new DeuxRaquettes();
        Balle b = new Balle();
        new Equipe(3,dr,b).start();
        new Equipe(3,dr,b).start();
    }

    public static void main(String[] args) {
        new InterblocageSemaphore(2,2);
    }
}
```

Exemple d'interblocage

```
class DeuxRaquettes {  
    Semaphore raquettes;  
  
    public DeuxRaquettes() {  
        raquettes = new Semaphore(2,true);  
    }  
  
    public void prendreRaquettes() {  
        try {  
            raquettes.acquire(2);  
            System.out.println("Equipe "+Thread.currentThread().getId()  
                +" : a pris les 2 raquettes");  
        } catch(Exception e) {}  
    }  
  
    public void rangerRaquettes() {  
        raquettes.release(2);  
        System.out.println("Equipe "+Thread.currentThread().getId()  
            +" : a rangé les 2 raquettes");  
    }  
}
```

```
class Balle {  
    Semaphore laBalle;  
  
    public Balle() {  
        laBalle = new Semaphore(1); // mutex  
    }  
  
    public void prendreBalle() {  
        try {  
            laBalle.acquire();  
            System.out.println("Equipe "+Thread.currentThread().getId()  
                +" : a pris la balle");  
        } catch(Exception e) {}  
    }  
  
    public void rangerBalle() {  
        laBalle.release();  
        System.out.println("Equipe "+Thread.currentThread().getId()  
            +" : a rangé la balle");  
    }  
}
```

```

class Equipe extends Thread {

    int nbmatchs;
    DeuxRaquettes raq;
    Balle balle;
    Random alea;
    public Equipe(int nbmatchs, DeuxRaquettes raq, Balle balle) {
        alea = new Random(System.currentTimeMillis());
        this.nbmatchs = nbmatchs;
        this.raq = raq;
        this.balle = balle;
    }

    public void run() {
        int match = 0;
        while(match < nbmatchs) {
            // on va chercher aléatoirement soit les raquettes, soit la balle
            if(alea.nextBoolean()) {
                raq.prendreRaquettes();
                balle.prendreBalle();
            } else {
                balle.prendreBalle();
                raq.prendreRaquettes();
            }
            jouer(match);
            balle.rangerBalle();
            raq.rangerRaquettes();
            match++;
        }
        System.out.println("===== Equipe "+getId()+" a terminé tout ses matchs");
    }

    private void jouer(int match) {
        try {
            System.out.println("Equipe "+getId()+" : joue match "+match);
            sleep(alea.nextInt(2000));
        } catch(InterruptedException ie) {}
    }
}

```

Exemple d'interblocage

Equipe 9 : a pris la balle
 Equipe 9 : a pris les 2 raquettes
 Equipe 9 : joue match 0
 Equipe 9 : a rangé la balle
 Equipe 10 : a pris la balle
 Equipe 9 : a rangé les 2 raquettes
 Equipe 10 : a pris les 2 raquettes
 Equipe 10 : joue match 0
 Equipe 10 : a rangé la balle
 Equipe 10 : a rangé les 2 raquettes
 Equipe 9 : a pris la balle
 Equipe 9 : a pris les 2 raquettes
 Equipe 9 : joue match 1
 Equipe 9 : a rangé la balle
 Equipe 9 : a rangé les 2 raquettes
 Equipe 9 : a pris les 2 raquettes
 Equipe 10 : a pris la balle

Eviter les interblocages (1/2)

A la conception

- une manière d'éviter les interblocages peut parfois être mise en oeuvre au moment de la conception
- si cela est possible, imposer l'ordre d'accès aux ressources permet de résoudre certains problèmes
- avec l'exemple de T_1 , T_2 , A et B :
 - 1 T_1 obtient le permis de la ressource A
 - 2 T_2 est bloqué au moment du `acquire()` sur la ressource A
 - 3 T_1 obtient donc le permis de la ressource B
 - 4 T_1 exécute son code
 - 5 T_1 relâche les ressources A et B et libère ainsi T_2

Eviter les **interblocages** (2/2)

Primitives dédiées

- imposer l'ordre d'accès aux ressources n'est pas toujours possible alors...
- pour éviter certaines situations il est possible d'invoquer une méthode **non bloquante** :
→ `tryAcquire()` qui retourne un booléen indiquant si un permis est disponible ou non.
- il est aussi possible d'attendre un permis durant un certain temps : `tryAcquire(durée, TimeUnit.MILLISECONDS)` (ou HOURS, MINUTES, SECONDS, etc.)
- il est aussi possible d'attendre plusieurs permis durant un certain temps : `tryAcquire(nb permis, durée, TimeUnit)`

Exercice

- revenons à notre problème des équipes qui veulent se téléphoner
- chaque équipe est un thread et chaque équipe a besoin de deux téléphones pour permettre à ses membres de communiquer
- chaque téléphone est un objet partagé dont l'accès est géré par un **mutex**

Énoncé du problème et contraintes

- on a k téléphones et n équipes
- chaque équipe commence par essayer d'obtenir un accès à l'un des téléphones choisi aléatoirement
- en utilisant certaines des primitives proposées par la classe Semaphore, proposez une solution pour ce problème qui évite les situations d'interblocage



Différence entre séaphore et *synchronized*

- un **séaphore** permet de limiter l'accès à une ressource à **un nombre maximum de threads** (éventuellement supérieur à 1)
- une section de code gérée par ***synchronized*** permet de garantir l'intégrité des données partagées et **limite l'accès à un seul thread**
- les deux mécanismes peuvent être combinés.

Exercice

Stationnement

- reprenez l'exercice du parking pour qu'il soit possible de savoir quelles sont les voitures qui l'occupent (via une ArrayList)

Références

- Edsger Wybe Dijkstra sur Wikipedia :
https://fr.wikipedia.org/wiki/Edsger_Dijkstra
- Présentation du mécanisme des sémaphores sur Wikipedia :
[https://fr.wikipedia.org/wiki/Sémaphore_\(informatique\)](https://fr.wikipedia.org/wiki/Sémaphore_(informatique))
- Le courrier de Dijkstra décrivant le mécanisme du sémaphore :
<http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD703.PDF>
- documentation Sémaphore :
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>
- Guide programmation concurrente en java :
<https://docs.oracle.com/javase/tutorial/essential/concurrency/>