Activité passager
// variables locales à l'activité
famille : {Montaigu, Capulet} := ... // Famille de l'activité
lanceur : booléen := false
// Sera-t-il celui qui démarre le bac ?
boucle
// code de synchronisation à compléter (pour pouvoir embarquer et
// éventuellement positionner lanceur)
prendre_place();
// code de synchronisation à compléter (pour s'assurer que tous sont
// sont à bord et éventuellement positionner lanceur)
si lanceur alors démarrer(); finsi;
// code de synchronisation à compléter (pour permettre un voyage ultérieur)
.
.
.
// débarquement et retour (hors sujet)
fin boucle

// Déclaration des sémaphores
Sémaphore mutex = new Sémaphore(1); // Sémaphore pour l'exclusion mutuelle
Sémaphore plein = new Sémaphore(0); // Sémaphore pour attendre que le bac soit plein
Sémaphore lanceurSem = new Sémaphore(1); // Sémaphore pour garantir l'exclusivité du lanceur

Activité passager
// variables locales à l'activité
famille : {Montaigu, Capulet} := ... // Famille de l'activité
lanceur : booléen := false
// Sera-t-il celui qui démarre le bac ?

boucle
// Code de synchronisation pour embarquer
mutex.down(); // Entrée en section critique
prendre_place(); // Embarquement
mutex.up(); // Sortie de la section critique

// Code de synchronisation pour s'assurer que tous sont à bord
plein.down(); // Attente que le bac soit plein
mutex.down(); // Entrée en section critique
if (lanceur) {
lanceurSem.down(); // S'assurer que seul le lanceur peut démarrer
démarrer(); // Démarrage du bac
lanceurSem.up(); // Libérer le droit de démarrer pour d'autres voyages
}
mutex.up(); // Sortie de la section critique

// Code de synchronisation pour permettre un voyage ultérieur
// (c'est-à-dire libérer le sémaphore pour le prochain voyage)
plein.up(); // Libérer une place pour le prochain voyage

// Débarquement et retour (hors sujet) fin boucle

Partie moniteur
Bonjour,

La méthodologie est bien suivie mais il y a deux pièges.

A) Condition d'acceptation pour embarquer(f) : ok si
- 1 ou 2 personnes embarquées (quelque soit leur famille)
- ou 3 personnes de la même famille et f = cette famille

- ou 1 personne de la famille f et 2 personnes de la famille complémentaire

Le premier cas permet d'atteindre 3 personnes quelconques qui peuvent être constituées en 3/0, 0/3, 2/1 ou 1/2. Les deux derniers cas sont pour respecter la contrainte 4/0 ou 2/2.

Il faut observer qu'il y a des conditions d'acceptation différentes pour C et M (cas 3/0 vs 0/3 et 1/2 vs 2/1), c'est comme si on avait deux opérations embarquerC et embarquerM. => il faut donc 2 variables conditions PeutEmbarquerC et PeutEmbarquerM.

Enfin, quand la 4eme personne embarque (supposons C), il faut qu'elle débloque les 3 autres en faisant attention que ça peut être PeutEmbarquerC.signal * 3 (cas où nbC = 4, nbM=0), ou PeutEmbarquerC.signal + PeutEmbarquerM.signal * 2 (cas nbC=2, nbM=2). C'est aussi cette personne qui aura la valeur de retour true.

B) Pour piloter, il faut utiliser le paramètre lanceur : on doit attendre si on n'est pas le lanceur et que le bateau ne bouge pas déjà. Il faut une variable d'état bateauLancé, la condition d'acceptation est alors : lanceur \/ bateauLancé.
Une variable condition nommée Piloter ou mieux DépartBateau.
L'activité qui a lanceur à vrai positionne BateauLancé à vrai et fait 3 DépartBateau.signal pour débloquer les 3 activités qui attendent le départ (il est possible que certaines ne soient pas encore arrivées à DépartBateau.wait, pas grave, le signal est sans effet).

C) La solution est ok avec priorité au signalé, elle ne l'est pas avec priorité au signaleur. Dans ce cas, on peut avoir, dans embarquer, une 4e personne qui arrive et signale 3 autres. Puis, avant que celles-ci ne soient effectivement débloquées, une 5e personne (de la même famille que la 4e) arrive et trouve aussi le prédicat vrai => elle passe et il y a 5 personnes dans le bateau.

Exécution en Exclusion Mutuelle (Mutex) :

- *Intérêt :* La simplicité est le principal avantage. En excluant mutuellement l'accès à toutes les variables partagées, on garantit la cohérence séquentielle.
- *Inconvénient :* La principale limitation est la potentialité de contention. Si plusieurs séquences de code tentent d'accéder simultanément aux variables partagées, cela peut entraîner des blocages et une perte d'efficacité.

2. **Verrous Spécifiques pour Chaque Variable :**

   - *Intérêt :* Réduit la contention en permettant à plusieurs séquences de code de travailler simultanément sur différentes variables. Améliore la parallélisme.
   - *Inconvénient :* Complexité accrue. La gestion des verrous spécifiques à chaque variable peut entraîner des risques d'ordonnancement et de bugs liés aux oublis de verrouillage ou aux inversions de priorité.

3. **Implémentation Non-Bloquante :**

   - *Intérêt :* Élimine complètement les blocages en permettant aux séquences de code de lire les variables partagées sans se bloquer mutuellement. Favorise un meilleur parallélisme.
   - *Inconvénient :* L'implémentation non-bloquante peut être plus complexe à concevoir et à tester. La gestion de la cohérence peut devenir délicate, et la complexité algorithmique peut augmenter.

4. **Mémoire Transactionnelle :**

   - *Intérêt :* Les transactions offrent une abstraction élevée qui facilite la conception. Elles garantissent la cohérence des séquences de code sans avoir besoin de spécifier manuellement les verrous.

   - *Inconvénient :* La surcharge liée à la gestion des transactions peut entraîner une perte de performance, en particulier dans des scénarios où de nombreuses petites transactions sont exécutées fréquemment. De plus, la complexité liée à la détection et à la gestion des conflits peut être élevée.

Le choix entre ces options dépend des caractéristiques spécifiques de l'application, telles que la fréquence d'accès aux variables partagées, la taille des transactions, la fréquence des écritures, et la complexité tolérée. En général, il est recommandé de profiler et de tester différentes approches pour déterminer celle qui convient le mieux à un cas d'utilisation particulier.

2 **Solution 1 (standard)**

```
Semaphore PE := new Semaphore(0); // événement ”Peut Entrer”
Semaphore PS := new Semaphore(0); // événement ”Peut Sortir”
Semaphore mutex := new Semaphore(1);
integer nbAttE := 0; // nb d’usagers en attente pour entrer
integer nbAttS := 0; // nb d’usagers en attente pour sortir
integer nbDans := 0; // nb d’usagers dans le bâtiment
Entrer() {
mutex.P();
si nbDans = 0
 ∧
nbAttE + nbAttS = 0 alors
nbAttE++; mutex.V(); PE.P();
sinon
si nbAttS = 1 alors // priorité aux sorties, pour la vivacité
nbAttS := 0 ; PS.V() ;
sinon
si nbAttE = 1 alors
nbAttE := 0 ; nbDans := 2 ; PE.V();
sinon
nbDans++ ;
fsi ;
fsi ;
mutex.V();
fsi
}
Sortir() {
mutex.P();
si nbDans = 2
 ∧
nbAttE + nbAttS = 0 alors
nbAttS++; mutex.V(); PS.P();
sinon
si nbAttS = 1 alors // priorité aux sorties, pour la vivacité
nbAttS := 0 ; nbDans := 0 ; PS.V() ;
sinon
// cas nbAttE= 1 impossible : si le batiment est non vide, on peut entrer directement
nbDans-- ;
fsi ;
mutex.V();
fsi
}
```

```
class Counter {
   private int value = 0;
```

```
    public synchronized void increment() {
        value++;
    }

    public int getValue() {
        return value;
    }
}

class Activity1 extends Thread {
    private Counter counter;

    public Activity1(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            counter.increment();
            System.out.println("Activity 1: Counter = " + counter.getValue());
        }
    }
}

class Activity2 extends Thread {
    private Counter counter;
    private Activity1 activity1;

    public Activity2(Counter counter, Activity1 activity1) {
        this.counter = counter;
        this.activity1 = activity1;
    }

    public void run() {
        // Wait for Activity1 to finish its work
        try {
            activity1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        for (int i = 11; i <= 20; i++) {
            counter.increment();
            System.out.println("Activity 2: Counter = " + counter.getValue());
        }
    }
}

class Activity3 extends Thread {
    private Counter counter;
```

```
    public Activity3(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        while (counter.getValue() < 20) {
            System.out.println("Activity 3: Counter = " + counter.getValue());
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Activity1 activity1 = new Activity1(counter);
        Activity2 activity2 = new Activity2(counter, activity1);
        Activity3 activity3 = new Activity3(counter);

        activity1.start();
        activity2.start();
        activity3.start();
    }
}
```

```
Un automate pur serait

0/0 --> 1/0 --> 2/0 --> 3/0 --> 4/0 --> démarrer
 |       |       |
 v       v       v
0/1 --> 1/1 --> 2/1
 |       |       |
 v       v       v
0/2 --> 1/2 --> 2/2 --> démarrer
 |
 v
0/3
 |
 v
0/4 démarrer
```

Exclusion mutuelle dans les moniteurs :

L'exclusion mutuelle dans les moniteurs est une garantie que seule une seule tâche à la fois peut exécuter une section critique du code à l'intérieur du moniteur. Cette garantie est assurée par le moniteur lui-même et est implémentée à travers l'utilisation d'une section critique protégée par un verrou (ou sémaphore).

L'exclusion mutuelle est nécessaire pour éviter les conditions de concurrence où plusieurs tâches peuvent accéder et modifier les mêmes données partagées simultanément. Si plusieurs tâches peuvent exécuter des sections critiques simultanément, des problèmes tels que les courses critiques peuvent survenir, entraînant des résultats indéterminés et des résultats incorrects.

Variables de condition dans les moniteurs :

Les variables de condition dans les moniteurs sont utilisées pour permettre la communication entre les différentes tâches qui utilisent le moniteur. Elles sont utilisées pour signaler et attendre des événements spécifiques, permettant ainsi une synchronisation plus fine entre les tâches. Les variables de condition permettent de mettre une tâche en attente jusqu'à ce qu'une certaine condition soit satisfaite, puis de la réveiller lorsque cette condition est remplie.

Rôle combiné de l'exclusion mutuelle et des variables de condition :

L'exclusion mutuelle et les variables de condition sont souvent utilisées ensemble dans les moniteurs. L'exclusion mutuelle protège les sections critiques du code contre l'accès simultané, tandis que les variables de condition permettent aux tâches de se signaler mutuellement lorsqu'un certain état est atteint, évitant ainsi la nécessité de boucles d'attente active.

Nécessité de l'exclusion mutuelle :

L'exclusion mutuelle est nécessaire pour éviter les courses critiques et garantir la cohérence des données partagées. Même si les variables de condition facilitent la synchronisation, elles ne fournissent pas par elles-mêmes une garantie d'exclusion mutuelle.

Peut-on résoudre tout problème avec une seule variable de condition ?

Non, on ne peut pas résoudre tous les problèmes de synchronisation avec une seule variable de condition. Une seule variable de condition peut être utile pour certains problèmes, mais elle n'est pas suffisante pour tous. Certains problèmes nécessitent la coordination de plusieurs événements ou conditions, ce qui peut nécessiter l'utilisation de plusieurs variables de condition pour décrire et gérer les différentes situations de synchronisation. Avoir une seule variable de condition peut conduire à des solutions limitées et complexes pour certains types de problèmes.