Rapport TP 2 Prog avancée :

1. Introduction :

Un sémaphore est un moyen de contrôler l'accès à une ressource partagée pour éviter les conflits et optimiser l'utilisation d'un programme. Il s'agit d'une structure de contrôle qui utilise une variable entière (integer) pour réguler l'accès aux ressources critiques. Il existe deux types de sémaphores :

- **Un sémaphore binaire** ne peut prendre que deux valeurs (0 et 1) et se comporte donc comme un verrou. Il est particulièrement utile lorsque l'accès exclusif à une ressource est requis. Lorsqu'un thread entre dans une section critique, il diminue la valeur du sémaphore à 0, empêchant ainsi les autres threads d'accéder à la ressource jusqu'à ce qu'il quitte la section critique et libère le sémaphore.
- **Un sémaphore général** peut prendre des valeurs supérieures à 1 ce qui permet de définir un nombre maximum de threads pouvant accéder simultanément à une ressource. Ce type de sémaphore est utilisé pour des ressources qui peuvent gérer plusieurs accès concurrents.

Le but de ce TP est donc de comprendre et appliquer l'utilisation de sémaphores dans un affichage et dans le précédent TP sur les mobiles.

2. Le fonctionnement des sémaphores :

Les sémaphores utilisent deux opérations :

- **Wait() (ou P) :** Cette opération diminue la valeur du sémaphore. Si la valeur devient négative, cela signifie que des threads sont en attente de la ressource.
- **Signal() (ou V) :** Cette opération augmente la valeur du sémaphore, libérant la ressource et permettant à un thread en attente d'y accéder.

En java, les sémaphores permettent de garantir qu'un seul thread à la fois puisse entrer dans une section critique.

3. Détails des classes pour affichage :

1. `TP_2/Main.java`

- **But** : Cette classe est le point d'entrée de l'application.
- **Utilisation du sémaphore** :
    - Un objet **`SemaphoreBinaire`** est créé pour gérer l'accès à une section critique entre les différents threads **(TA, TB, TC, et TD)**.
    - Chaque thread suit un cycle structuré :

        - Il appelle **`syncWait`** avant d'entrer dans la section critique, ce qui diminue la valeur du sémaphore. Si cette valeur est déjà à 0, le thread attend son tour.
        - Une fois dans la section critique, le thread effectue une tâche d'affichage de texte.
        - Après avoir terminé, le thread appelle **`syncSignal`** pour libérer la section critique, ce qui permet à un autre thread d'y accéder.

2. `TP_2/NewAffichage.java`

- **But** : Cette classe représente un thread responsable de l'affichage de texte, affichant chaque caractère du texte de manière séquentielle et avec un léger délai.
- **Utilisation du sémaphore** :
    - Bien que la classe elle-même n'interagisse pas directement avec le sémaphore, les instances de **`NewAffichage`** sont créées et gérées par les threads dans **`Main.java`**. Grâce à l'exclusion mutuelle assurée par le sémaphore dans **`Main.java`**, chaque instance de **`NewAffichage`** affiche son texte dans un environnement synchronisé, évitant les conflits d'accès.

3. `TP_2/Affichage.java`

- **But** : Cette classe représente un autre type de thread chargé d'afficher du texte, mais avec une gestion explicite de l'exclusion mutuelle lors de l'affichage.
- **Utilisation du sémaphore** :
    - Contrairement à **`NewAffichage`**, cette classe utilise un objet **`Exclusion`** pour garantir une exclusion mutuelle explicite. Cela signifie que même si plusieurs instances de **`Affichage`** sont lancées, le verrou d'exclusion mutuelle garantit que seul un thread à la fois peut effectuer l'affichage du texte.

4. `TP_2/SemaphoreBinaire.java`

- **But** : Cette classe représente un **sémaphore binaire**, c'est-à-dire un sémaphore qui ne peut prendre que les valeurs 0 ou 1.
- **Utilisation du sémaphore** :
    - **`SemaphoreBinaire`** étend la classe abstraite **`Semaphore`** pour créer un sémaphore binaire spécifiquement destiné à limiter l'accès à une ressource unique.
    - La méthode **`syncSignal`** est redéfinie pour s'assurer que la valeur du sémaphore ne dépasse jamais 1, maintenant ainsi l'exclusivité de la ressource partagée.

- Ce sémaphore binaire joue donc un rôle essentiel en empêchant tout accès concurrent à la section critique lorsque la ressource est déjà utilisée par un thread.

5. `TP_2/Semaphore.java`

- **But** : Classe abstraite de base fournissant les méthodes essentielles pour un sémaphore, comme **`syncWait`** et **`syncSignal`**.
- **Utilisation du sémaphore** :
  - `Semaphore` contient la logique centrale de synchronisation, gérant l'état du sémaphore à travers des méthodes synchronisées.
  - La méthode **`syncWait`** réduit la valeur du sémaphore, indiquant qu'un thread utilise la ressource. Si la valeur est 0, le thread est mis en attente.
  - La méthode **`syncSignal`** augmente la valeur du sémaphore, libérant la ressource pour qu'un autre thread puisse l'utiliser.
  - Cette structure permet aux classes dérivées, comme **`SemaphoreBinaire`**, de réutiliser ces fonctionnalités tout en adaptant le comportement selon les besoins (par exemple, en limitant la valeur à 1 pour un sémaphore binaire).

4. Comparaison avec le TP 1 :

Dans TP_2/UnMobile.java, les sémaphores jouent un rôle essentiel en régulant l'accès à la section centrale où les mobiles se déplacent. Grâce à ce mécanisme, on s'assure qu'un seul mobile à la fois peut entrer dans cette zone critique, évitant ainsi tout risque de collision.

Points Clés :

- **Déclaration du Sémaphore :** Un SemaphoreGeneral statique est défini pour être accessible à toutes les instances de UnMobile, ce qui en fait un outil central de gestion de l'accès à la zone critique.
- **Utilisation du Sémaphore :**
  - **syncWait :** Cette fonction est appelée avant qu'un mobile entre dans la section centrale. Si un autre mobile y est déjà, le mobile actuel attend son tour.
  - **syncSignal :** Une fois qu'un mobile a traversé la section centrale, il appelle cette fonction pour indiquer que la zone est désormais libre.

Dans TP_1/UnMobile.java, il n'y a pas d'utilisation de sémaphores. Les mobiles se déplacent librement d'un point A à un point B sans qu'aucun mécanisme ne régule leur passage dans une zone spécifique. Cela signifie qu'ils n'ont pas de synchronisation pour éviter les conflits d'accès.

Points Clés :

- **Pas de Sémaphore :** Aucun sémaphore n'est utilisé pour limiter l'accès des mobiles à une zone spécifique de leur mouvement.
- **Mouvement Simple :** Les mobiles se déplacent d'un bout à l'autre sans se préoccuper de la présence des autres, ce qui rend le mouvement plus basique et moins contrôlé.

Pour résumer :

- **TP_2/UnMobile.java :** Utilise les sémaphores pour s'assurer qu'un seul mobile à la fois occupe la section centrale, offrant ainsi une gestion efficace et sécurisée du mouvement.
- **TP_1/UnMobile.java :** Se passe de sémaphores, laissant les mobiles se déplacer sans restriction, ce qui simplifie l'implémentation mais ne permet pas de gérer les accès simultanés de manière sécurisée.

5. Conclusion :

Ce TP a permis de mieux comprendre et comment appliquer des concepts de synchronisation et d'exclusion mutuelle via les sémaphores, éléments cruciaux dans la gestion des ressources partagées. Nous avons étudié les sémaphores binaires et généraux, ainsi que leurs implémentations pour mieux contrôler l'accès aux sections critiques dans un environnement multi-threadé. Les différentes classes implémentées illustrent l'importance de la synchronisation dans l'affichage séquentiel de données et la prévention des conflits d'accès. De plus, cela nous a démontré l'efficacité des sémaphores pour assurer la cohérence des opérations concurrentes, un savoir essentiel pour optimiser et sécuriser le fonctionnement des programmes multithreadés.