
**TP**

—

# Compte rendu TP
# Programmation avancée

Réalisé par :

MOUSSAMIH Elias

—

INF3 - FA

Composants de l’ordinateur

Intel i7-12700 T 3.4GHz 12 coeurs / 20 threads
64Go ram
SSD 480 Go
Windows 11 professionnel education

Des IA ont été utilisées pour comprendre plusieurs notions, ces notions ont par la suite été expliquées par moi même sans l’aide d’IA pour rédiger.

# TP1

---

Lorsque le programme est lancé, un thread est créé pour faire fonctionner l'objet UnMobile. Celui-ci déplace un rectangle d'un côté à l'autre de la fenêtre en appelant périodiquement repaint() pour actualiser l'affichage.

On crée d'abord le Mobile en donnant sa Hauteur et sa Largeur. Quand le Mobile est créé il faut ensuite l'ajouter à la fenêtre en faisant :

add(NomMobile);

Une fois le Mobile ajouté, on crée maintenant le Thread associé en donnant en paramètre le nom du mobile lors de la création :

Thread thread = new Thread(NomMobile);

On démarre maintenant le thread avec thead.start() puis on rend le tout visible avec setVisible(true)

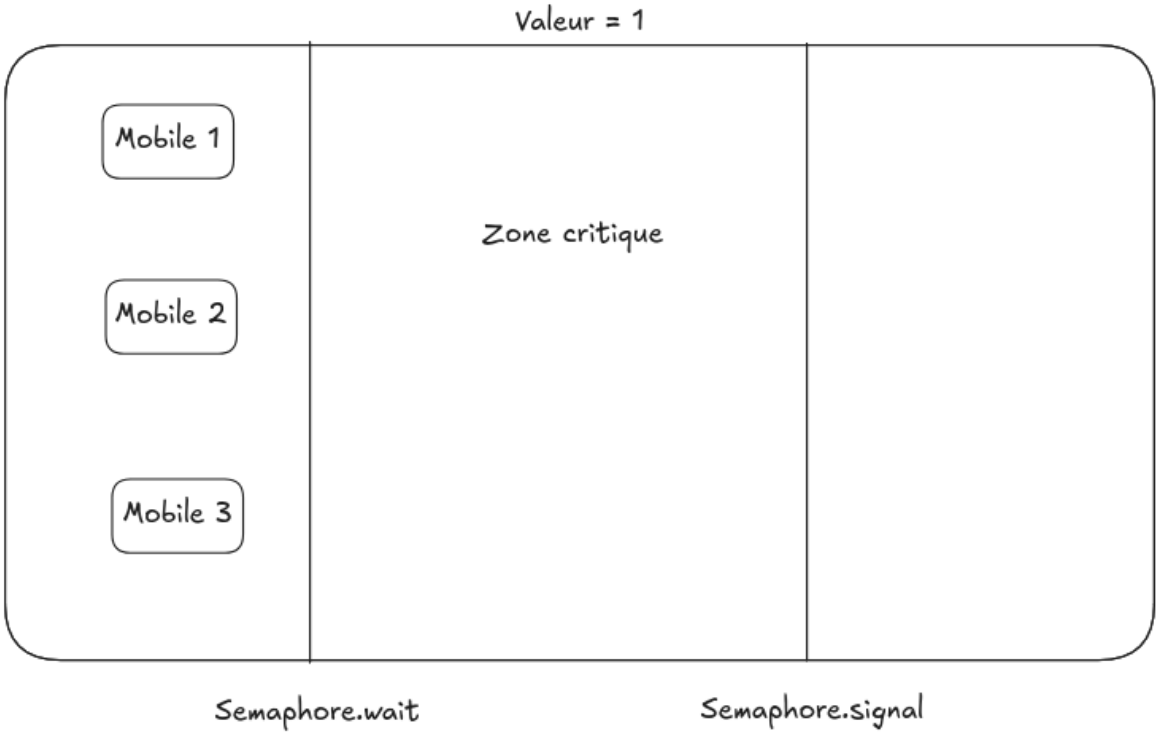
On utilise paintComponent(Graphics) pour recréer le mobile au bon emplacement à chaque mouvement.

On ajoute maintenant un sémaphore pour gérer l'accès des threads en zone critique.

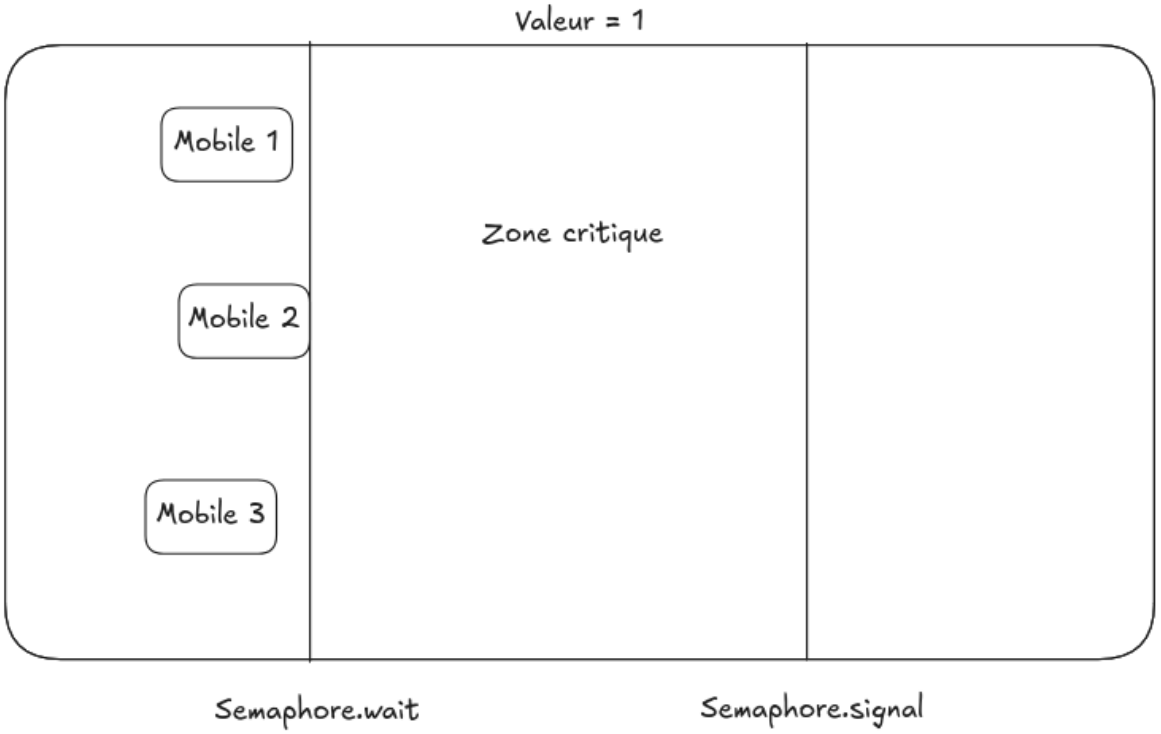
Les threads essaient tous les deux en même d'accéder à la sortie standard.

Tous les threads arrivent et le sémaphore s’occupe de n’en laisser passer qu’un seul avec wait.
Une fois que le thread a fini, il envoie un message au sémaphore et libère la ressource que le sémaphore laisse au thread suivant.

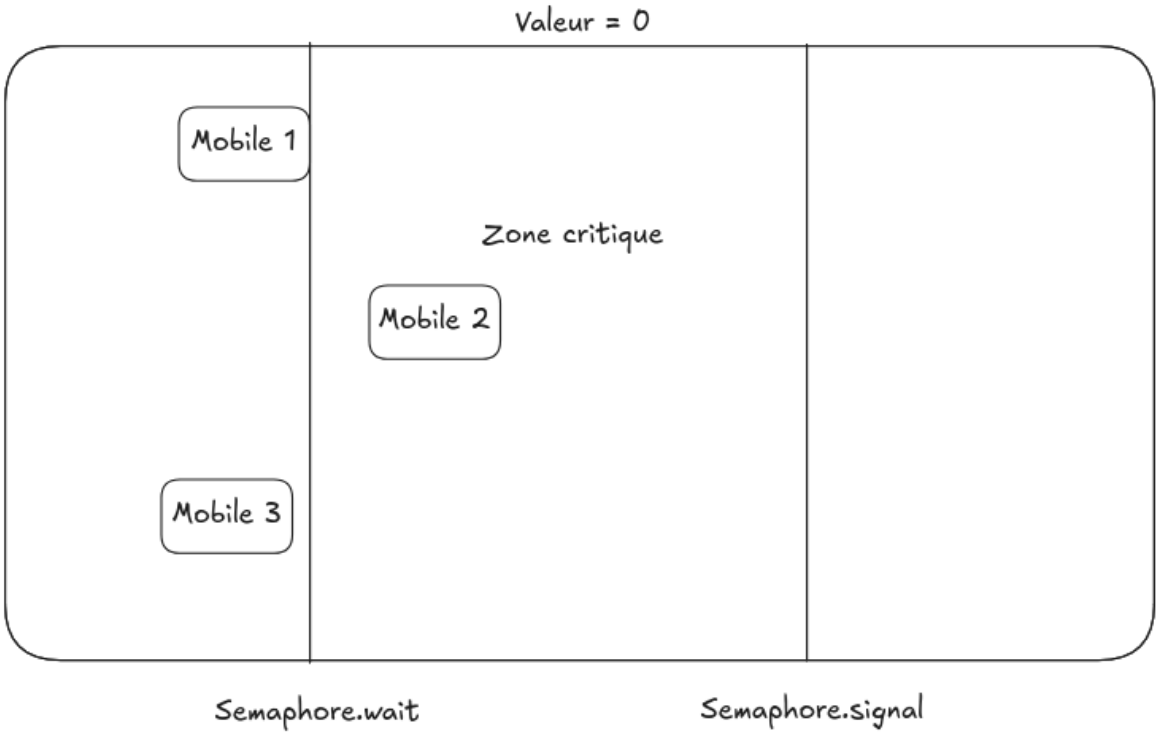
Pour que le thread comprenne qu’il ne peut pas passer, la valeur est à 0. Quand le thread fait signal, la valeur est changée à 1. A ce moment la, un nouveau thread est choisi par l’OS et quand il commence sa tâche il met la valeur à 0.



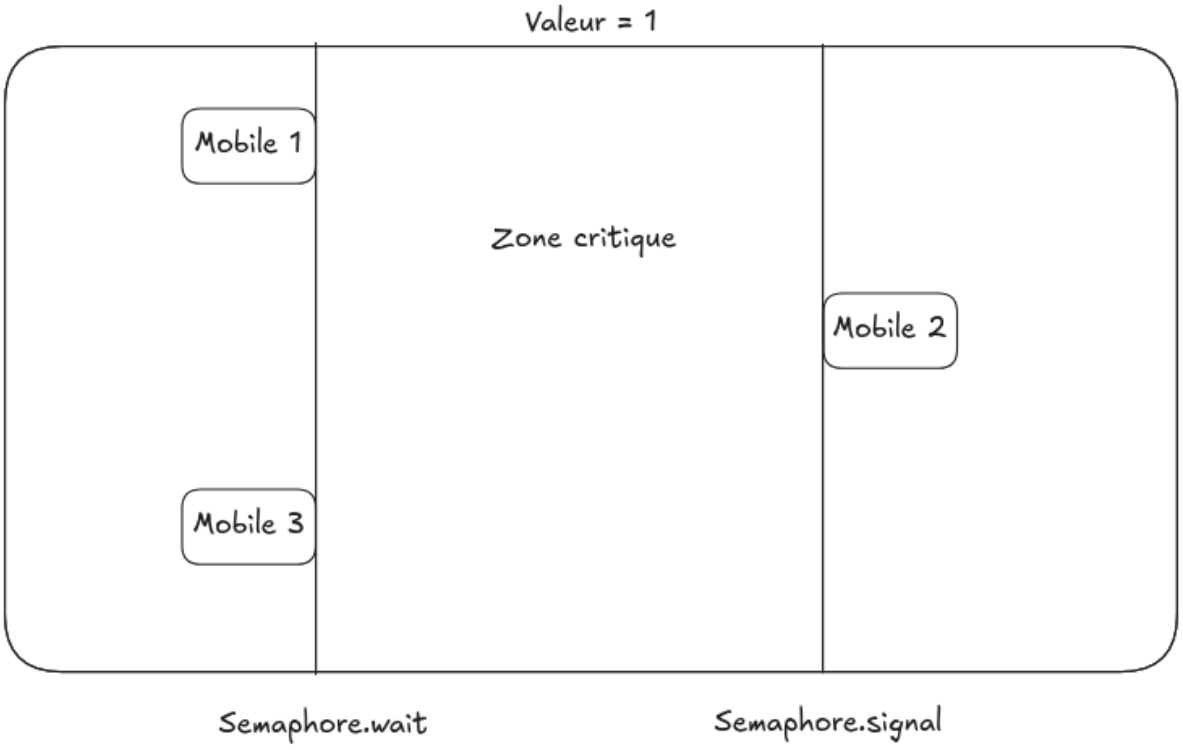

A ce moment, les mobiles n'ont pas encore atteint la zone critique. La valeur correspond au nombre de places disponibles pour les mobiles dans la zone critique.



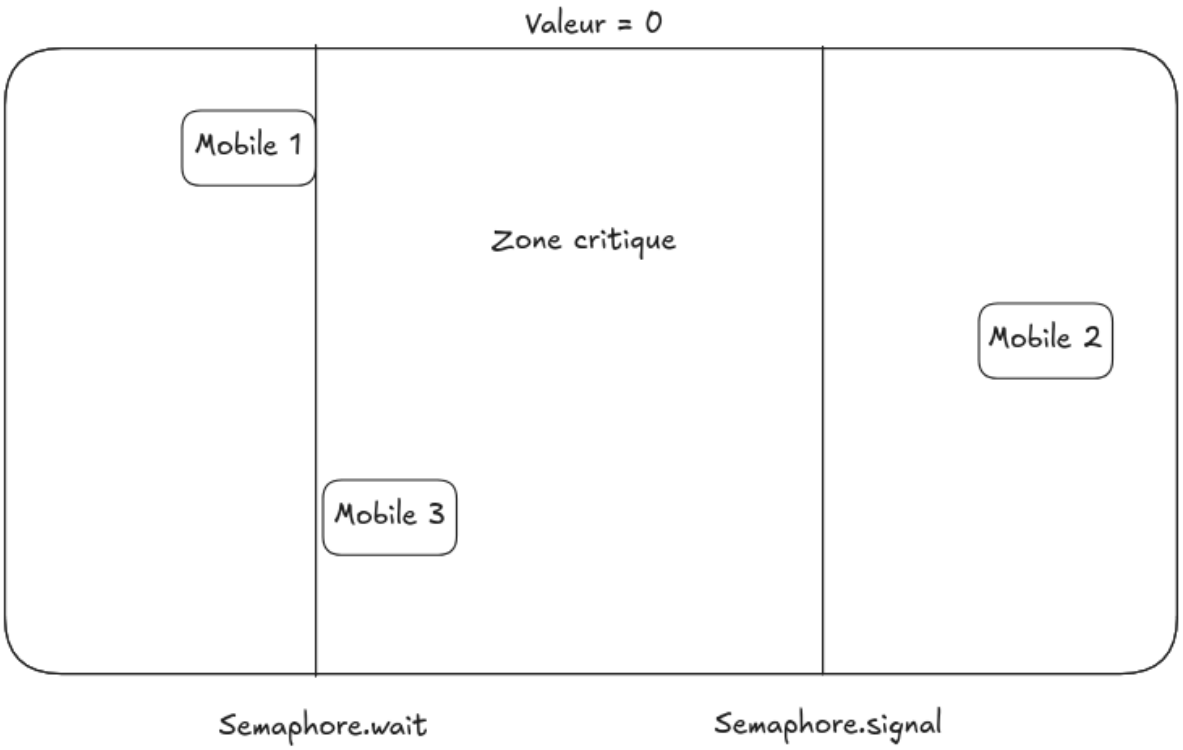
Le Mobile 2 arrive en zone critique, le Sémaphore regarde donc s' il y a déjà un mobile en zone critique via la variable Valeur. Puisqu'elle est actuellement égale à 1, le Sémaphore accepte que le Mobile 2 passe en zone critique.



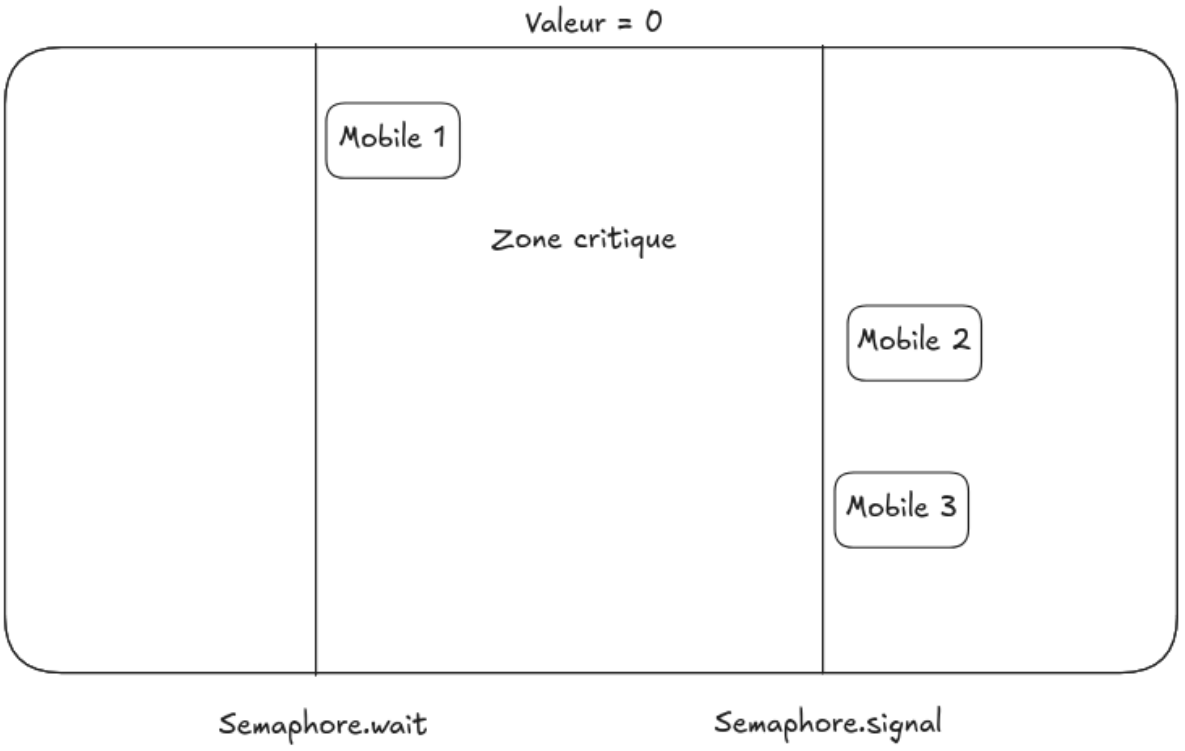

Le Mobile 2 est maintenant en zone critique, le Mobile 1 est arrivé à l'entrée de la zone critique mais le Sémaphore ne le laisse pas y entrer car le Mobile 2 y est déjà et a donc mis la Valeur à 0. Puisqu'elle est à 0, le Sémaphore sait qu'il y a un Mobile en zone critique.



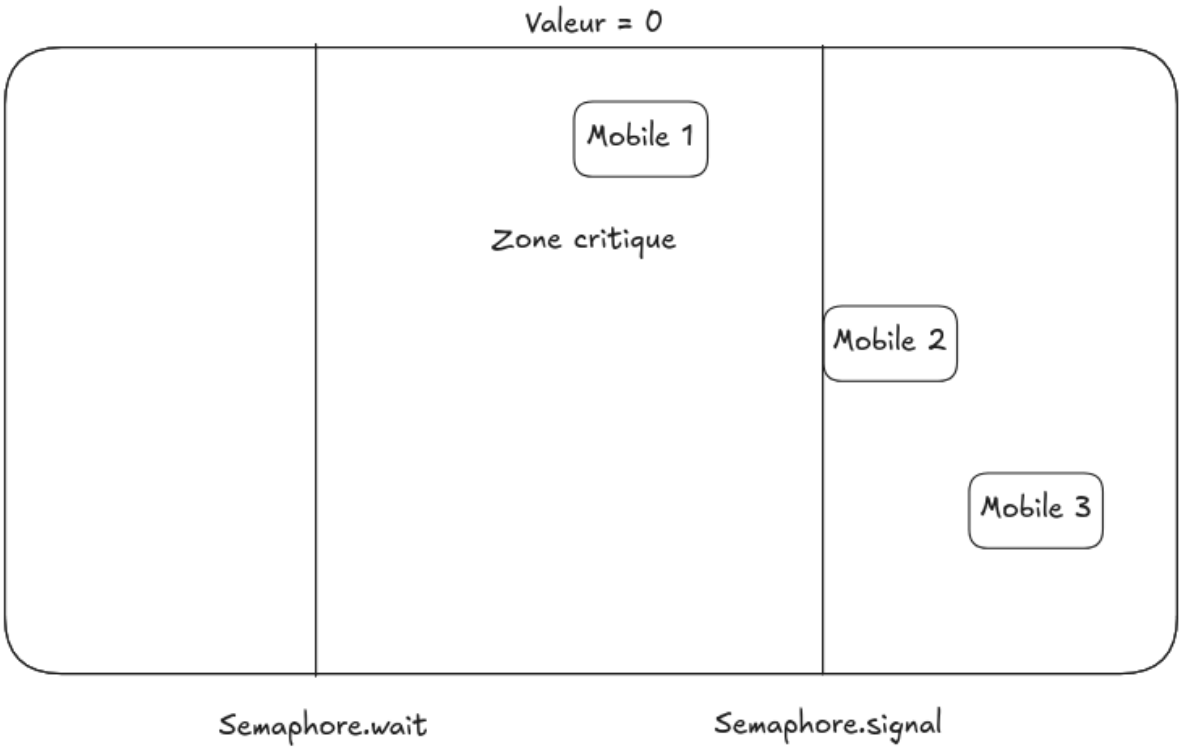

Il ne reste plus aucun mobile en zone critique, le mobile 2 qui en sort signale donc au Sémaphore qu'il en sort. Le valeur revient alors à 1 ce qui montre au Sémaphore qu'il peut laisser un nouveau mobile entrer en zone critique.



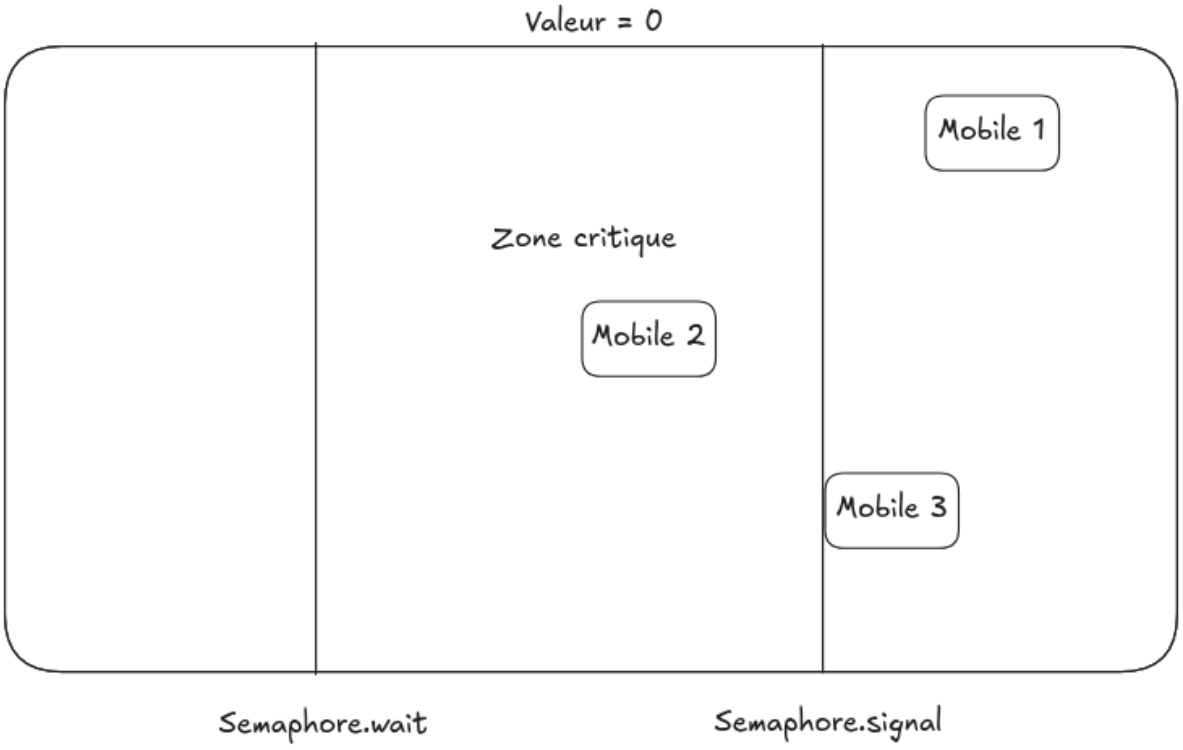
Même si le mobile 1 est entré en file d'attente en premier, il n'entrera pas en zone critique forcément en premier. Comme on peut le voir, le Mobile 3 est entré en zone critique avec le Mobile 1. La valeur est donc retournée à 0 puisqu'il y a un mobile en zone critique.




Une fois que le mobile 3 est sorti de la zone critique, le Mobile 1 peut à son tour y entrer. Le Mobile 2 à lui eu le temps d'arriver au bout de la page et de commencer à revenir.



Le mobile 2 arrive à l'entrée de la zone critique, même s' il n'avance pas dans le même sens que le mobile 1, il ne peut pas entrer en zone critique.




Une fois que le mobile 1 est sorti de la zone critique, le Mobile 2 peut enfin y entrer.

En résumé, grâce au sémaphore, les mobiles peuvent entrer en zone critique sans se gêner. Le sémaphore agit comme un gardien, qui laisse entrer un mobile à la fois en contrôlant la valeur. Tant qu'un mobile est dans la zone critique, la valeur est à 0, empêchant les autres d'y accéder.

Même si un mobile arrive en premier, cela ne garantit pas qu'il sera le prochain à entrer en zone critique. Comme on l'a vu, Mobile 3 a pu passer avant Mobile 1. Cela dépend de l'OS, qui décide quel mobile va entrer en premier en fonction de sa propre gestion des threads.

En conclusion, le sémaphore joue un rôle essentiel en permettant aux mobiles de se succéder en zone critique sans conflit. Chaque mobile attend son tour, et le système reste organisé et fluide grâce au sémaphore qui gère l'accès en fonction de la valeur.

## TP2

Maintenant les threads sont représentés par un affichage de chaîne de caractères. On crée un nouvel affichage avec :

Affichage NomAffichage = new Affichage("AAA")

On peut remarquer que l'ordre de création des affichages n'influe pas dans l'ordre de passage des threads.

On fait ensuite pour chaque thread NomAffichage.start()
L'ordre n'est toujours pas défini car il est attribué par l'OS.

## TP3

On représente maintenant les sémaphores via un exemple de boîte aux lettres. Dans ce cas, un Producteur utilise la méthode write() pour "déposer une lettre" le lecteur va lui utiliser read() pour lire cette lettre. Le sémaphore sert dans ce cas à faire en sorte qu'il n'y ait pas de conflit entre les éléments. Le producteur appelle donc syncWait() avant d'écrire une lettre puis appelle syncSignal() une fois qu'il a terminé et le lecteur fonctionne de la même manière. En d'autres termes, syncWait et syncSignal permettent de limiter le nombre de threads qui accèdent à la ressource partagée et leurs donnent également accès, une fois que c'est possible à ces ressources.

Les méthodes sont synchronisées afin de garantir l'accès à un seul thread, prévenir les erreurs et les comportements imprévisibles ainsi que les incohérences.

On représente un exemple de gestion de ressources partagées via une boulangerie. Dans ce cas, un boulanger utilise la méthode deposer() pour "ajouter un pain", tandis que le Client utilise la méthode acheter() pour "prendre un pain". Le sémaphore est utilisé ici pour garantir qu'il n'y ait pas de conflit entre les actions des clients et du boulanger.

Le boulanger appelle donc syncWait() avant de déposer un pain, puis syncSignal() une fois qu'il a terminé, et le client utilise le même processus pour s'assurer que l'accès au stock se fasse de manière ordonnée. En d'autres termes, syncWait() et syncSignal() limitent le nombre de threads accèdent au panier de la boulangerie et leur permettent d'accéder aux pains quand c'est possible.

Les méthodes acheter() et deposer() sont synchronisées afin de garantir l'accès à un seul thread à la fois, prévenir les erreurs, les comportements imprévisibles et assurer la cohérence du stock. Cela évite des situations où un client pourrait retirer un pain alors que le boulanger est en train de le déposer, ou des conditions de concurrence où plusieurs clients tenteraient d'acheter le même pain.