# Simulateur d’aéroport

Laboratoire 3, programmation concurrente, par Florian Fasmeyer et Cédric Pahud.

## Introduction

Le but de ce laboratoire est de simuler un aéroport en utilisant un thread pour chaque avion. Ceci doit être fait de deux manières différentes : en utilisant des blocking queues, puis en utilisant un système de tampon avec des blocks synchronized. Une fois les deux méthodes implémentés nous avons dû tester leurs temps d’exécutions pour en tirer des conclusions.

## Mode d’emploi

Le programme se lance en ligne de commandes (dans eclipse : Run / Run configurations / arguments / Program arguments). Les arguments sont dans l’ordre suivant : nombres d’avions, nombre de pistes d’atterrissages, nombre de pistes de départ, nombre de places sur le tarmac et usage de blocking queues (bool permettant de choisir la méthode). Donc si nous rentrons « 20 2 2 4 true » notre programme utilisera les blocking queues, aura 20 avions, 2 pistes de décollage, 2 pistes d’atterrissages et 4 places sur le tarmac.

## Code

AirportFrame.java contient les fonctions permettant à l’affichage de se rafraichir en fonction du code de nos threads. Dans Main.java l’on récupère le paramètre de la ligne de commande et en fonction de si l’on doit utiliser blocking queue ou non, l’on instancie les avions avec l’un des deux constructeurs différents (blocking queues ou simple listes), ensuite l’on start nos threads et on implémente ce que doivent faire les boutons stop et start (on utilise un sémaphore pour gérer cela). Dans Avion.java. une fonction isPaused() utilise une sémaphore (initialisée statiquement avant le main) afin de mettre en pause les avions.

## Tests

Voir fichier excel.

Les tests menés ne sont pas exhaustifs et n’ont été réalisés qu’une seule fois pour chacune des 24 catégories. Les données obtenues n’en restent pas moins informatives et nous permettent de faire de bonne (bien que dramatiques) observations.

## Conclusion

Le temps te traitement est approximativement même que le temps calculé (voirLabo3.xlsx). L’on peut observer entre les listes et les listes bloquantes une différence de ±100ms en faveur des listes bloquantes. L’on remarquera une différence importante entre le test no.4 de chaque type ; différence très probablement dû à un évènement extérieur aux tests.

Nous remarquons ainsi que les listes bloquantes sont légèrement plus rapides que les listes simples. Le résultat attendu étant à l’opposé de nos attentes, l’on en déduit que notre implémentation possède des zones bloquantes inutiles, des ‘bottleneck’ où l’on perd du temps. Il est fort probable que cette perte de temps soit due à notre usage de sémaphores afin d’implémenter la fonction pause. Les threads se mettent en attente dans la fonction isPaused(). L’usage d’un booléen (non synchronisé) aurait été plus judicieuse étant donné que la synchronisation de la pause n’est pas primordiale. Il n’y a pas besoin de synchroniser si l’on ne compte pas modifier. Tout le monde devrait pouvoir lire en même temps, une mise en pause légèrement retardée n’est pas un problème.