Projet LO41 : Aéroport

Rapport de conception

LE VAGUERÈS Yann

UTBM Département informatique
LO41 Semestre P20

Juin 2020

Table des matières

Avertissement

Les premières parties relèvent des structures utilisées durant le programme. Si vous voulez **directement passer à l'algorithme** et étudier **comment je gère les avions et l'attente**, vous pouvez directement **commencer** dans la partie mutex. Si vous souhaitez plus d'informations sur les **données**, vous pourrez toujours **revenir** aux parties précédentes. Si vous souhaitez vraiment **vous concentrer sur le plus principal**, lisez la partie sur le thread d'avion, le thread du contrôleur,la liste d'attente et la configuration du programme.

But du projet

Le projet suivant a pour but de **mettre en oeuvre** les **compétences acquises** lors du module de **LO41** sur le fonctionnement des systèmes d'exploitation au travers d'un **cadre pratique réel**, celui de la **gestion** des **avions** d'un **aéroport**. De plus, il met à l'épreuve notre faculté à **concevoir** des **algorithmes** permettant cette gestion. Ainsi, ce rapport permet d'**expliquer comment nous y sommes arrivés**.

Concepts utilisés

Dans ce projet, on utilisera les concepts suivants :

- Les threads
- Les mutex
- La mémoire partagée
- Des signaux

Cela paraît très court mais le projet ne nécessite pas vraiment plus. Il nous faut un moyen de **bloquer/libérer les avions**, **stocker des informations et les partager de manière sûre**. Pour coder proprement, on gère l'interruption du programme par l'utilisateur.

Des entités

Le projet s'ancre autour d' “entités”. Ces entités sont des **représentations structurées de données** d’objets concret : un **affichage** ligne, un affichage **colonne**, un **avion**, une **piste**, une zone de stationnement d’avion qu’on appellera par la suite un **tarmac**. Je définirai toutes ses entités de manière précise et comment elles **interagissent** entre elles.

Des structures, des structures partout

Vous allez vite vous en rendre compte en parcourant le programme mais afin de **créer**, **partager** des informations, et cela de manière **sûre**, j’ai décidé de créer des structures. Celles-ci permettent de **simplifier** l’**écriture** et la **lecture** du code et de **transmettre** de manière **logique** les informations **entre les différentes entités**.

Affichage

J’ai créé des structures pour l’affichage, permettant d’afficher les informations de manière organisée sur l’écran.

Structure

Un affichage en ligne, dit **affichage**, est représenté par les propriétés suivantes :

- Une **marge en haut**, une **marge bas** et une **marge gauche** qui permettront d’espacer le contenu à souhait
- Une **hauteur**, qui définit sur combien de lignes on peut écrire
- Un **précédent** et un **suivant**, qui sont des pointeurs d’affichages, vers les affichages voisins

Fonctionnement

Les **affichages** fonctionnent comme une **liste doublement chaînée**. Grâce aux différentes fonctions crées, on peut écrire dans un affichage de la même manière qu’on peut avec une instruction **printf** avec un **format** et des **arguments optionnels**. Comme toute liste chaînée, on peut **ajouter** ou **supprimer** des affichages. Ce chaînage permet de trouver le décalage vertical nécessaire pour afficher le contenu au bon endroit.

Colonne

Structure

Un affichage colonne, dit **colonne**, est une colonne au sein d'un **affichage**. Avec plusieurs **colonnes** dans un **affichage**, on peut représenter **plus de données sur une même ligne**. Il est représenté par les propriétés suivantes :

- Une **largeur**, un entier qui représente le nombre de caractères affichables
- Un chaîne de caractère qui stocke le **contenu**
- Un **précédent** et un **suivant**, qui sont des pointeurs de **colonnes,** vers les **colonnes** voisines
- Un pointeur vers l'**affichage** utilisé

Fonctionnement

Tout comme les **affichages**, les **colonnes** fonctionnent comme une **liste doublement chaînée**, seulement de manière horizontale. Des fonctions sont fournis afin d'initialiser tout un tableau de colonne pour un affichage donné. De manière analogue aux fonctions pour **affichage**, on peut **changer le contenu** avec *colonneChangeContenu* et **mettre à jour** le contenu sur l'écran grâce à *colonneUpdate*. De plus, on peut **mettre à jour une ligne de colonnes** avec la fonction *colonneLigneUpdate*. Ce chaînage permet de trouver le décalage horizontal nécessaire pour afficher le contenu au bon endroit.

Avion

Structure

Un avion est représenté par les propriétés suivantes :

- Un **numéro de vol**, une chaine de caractère qui permet à l'utilisateur de reconnaître l'avion
- Un **numéro**, un entier qui est unique à l'avion
- Un **lieu**, un pointeur de chaîne de caractère, qui désigne la provenance/destination de l'avion parmis le choix disponible
- Une variable binaire permettant de savoir si l'avion **arrive** ou pas

- Une quantité de **fuel** représentée par un nombre entier
- Une variable binaire permettant de savoir si l'avion est **gros** ou pas
- Une variable binaire permettant de savoir si l'avion doit effectuer un **atterrissage d'urgence**
- Un **temps,** qui indique la date et heure d'arrivée/départ de l'avion
- Une **date**, qui représente de manière humaine la propriété **temps**

Je ne détaillerai pas le fonctionnement de l'avion ici, il sera beaucoup plus pertinent d'en parler pendant le **thread d'avion** ou le **thread de contrôleur**.

Piste

Structure

La piste permet de faire atterrir les avions, et bien qu'il y ait peu d'informations, celles-ci sont essentielles à l'affichage des informations. Une piste est représentée par les propriétés suivantes :

- Un pointeur vers **l'avion en cours**, ainsi on peut récupérer ses informations de vol
- Une variable binaire disant si la piste et **longue** ou pas
- Un **affichage**, la vue associée à la piste qui sera affichée.

Fonctionnement

Il y aussi très peu de fonctions pour la piste, la principale étant *pisteAfficher*, qui permet d'afficher la piste. On peut ainsi voir l'occupation de la piste et des informations sur l'avion, notamment si c'est un atterrissage d'urgence.

Tarmac

Structure

La zone de stationnement, ou **tarmac**, est une structure essentielle à la gestion des avions de l'aéroport. En effet, celui-ci possède une taille, qui conditionne la **place disponible pour atterrir**. Un tarmac est représenté par les propriétés suivantes :

- Un **titre**, la chaîne de caractère qui sera affichée
- Un **affichage** pour le **titre**, choix uniquement d'optimisation

- Un **affichage** pour les colonnes
- Un tableau de **colonnes**, qui permettent d'afficher les places de libre
- Un tableau de pointeurs d'**avions**, qui stocke tous les avions actuellement garés. On se sert de ce champ afin de connaître le nombre de places disponibles à tout moment

Le tarmac n'a pas de fonctions autres que pour l'affichage. Cependant, ces **données** sont **essentielles** à la gestion de l'aéroport.

Arguments

Lors du démarrage des threads, on va faire **passer** nos **données** dans l'**argument**. J'ai donc **créé** des structures afin de rendre ça plus facile. Il existe **deux structures**, une pour le **contrôleur**, l'autre pour les **avions**. Les **arguments** sont créés dans le **programme principal**.

Celle pour le contrôleur possède les propriétés suivantes :

- Un pointeur vers le **mutex pour les avions** (expliqué dans la suite)
- Un entier correspondant à l'**identifiant** de la **file de message** créée

Celle d'un avion possède les propriétés suivantes :

- Un pointeur vers les **arguments** du **contrôleur**, afin d'avoir au moins les mêmes informations que le contrôleur
- Un entier correspondant à l'indice de l'avion, utile tout au long de l'exécution pour le contrôler

Des mutex

Des mutex pour les avions

Éléments et données critiques, il nous **faut** un **mutex pour les avions**. Grâce à ce mutex, on peut **contrôler l'atterrissage/décollage des avions** avec le **tableau de moniteurs** de taille du nombre d'avions et avec un autre **moniteur**, **informer d'un atterrissage forcé**, ou une **action quelconque** d'un avion effectué. À cela, on associe une **structure** afin de tout transporter, le mutex et les moniteurs associés.

Pas de mutex pour les pistes

On penserai que les **pistes** sont des **ressources critiques indépendantes** nécessitant d'être **sécurisées**, cependant je vais vous expliquer pourquoi il n'y en a pas. Ces explications sont aussi disponibles dans le fichier *mutex_pistes_struct.h*.

On aurait bien fait une **structure** avec un **mutex**, **deux moniteurs**, une pour **chacune piste**, et un dernier **moniteur** pour une **action quelconque sur une piste**. Cependant chacun de ces champs est **inutile**, et voilà pourquoi.

Le **mutex réserve l'accès** aux **conditions d'exécution** et des **ressources partagées**, mais quelles sont ces conditions et ressources partagées ? Les conditions seraient les moniteurs des pistes et le moniteur d'une action sur une piste.

Le moniteur *avionQuelconque* est inutile car **elle revient** au moniteur *avionQuelconque* du mutex pour les avions. En effet, quand il y a un atterrissage/décollage d'un avion, il y a **forcément** une piste de libérée.

Les moniteurs *conditionsPistes* ne nous servent à rien car **la piste n'a aucune volonté propre**, **nous le contrôleur** sommes le **seul** à autoriser le décollage / atterrissage des avions.

Des signaux

On implémente dans notre programme des **signaux**, et plus spécifiquement le **traitement** du signal de **terminaison** car notre programme est censé tourner à l'infini. Ainsi, on peut **libérer** par exemple les **avions**, les **files de messages**, ou encore terminer tous les **threads** lancés. Cependant, l'**allocation dynamique** a été **réduite** à son minimum grâce aux **variables de précompilation**.

Envoyer des messages

À la base, on se servait de **messages IPC** pour envoyer des **messages** entre les **threads**. Cependant les messages **IP**C sont **uniquement entre processus** donc ils sont **inutilisables** lorsqu'il s'agit de **communiquer** entre des **threads**. Un moyen aurait été de créer une file et de se servir d'un **mutex** afin d'en **protéger l'accès**. Finalement j'ai créé

un tableau de pointeurs d'avions et un compteur d'avions ajoutés. Quand le **nombre** d'avions ajouté est **égal** au **nombre total d'avions**, alors le dernier **avion** ajouté **notifie** le **contrôleur** qu'il connaît maintenant tous les avions.

Des threads

Le programme principal lance des threads. Ces threads, afin d'être plus simples, sont représentés par une structure. un thread possède les propriétés suivantes :

- Un **identifiant** de thread, un **thread_t**
- Un pointeur vers la fonction à lancer lors du démarrage du thread

Dans notre programme, on va créer un **thread** pour notre **contrôleur**, et un **thread** pour **chaque avion**. On va pouvoir les lancer grâce à la fonction *lancerThread* prenant comme argument un pointeur vers une **structure de thread** et un pointeur **argument**.

Le thread d'avion

Le thread d'avion est finalement très basique dans son fonctionnement. On considère l'avion comme "**infini**". Un avion n'est **jamais "détruit"** quand il décolle. En réalité, il est "**remplacé**" par un nouvel avion avec de **nouvelles propriétés** : une nouvelle provenance, un nouveau numéro de vol, un nouveau niveau de fuel et on sait s'il doit atterrir d'urgence, une nouvelle heure. Ce choix permet de **faciliter** la **création/destruction** des **avions** mais aussi **faciliter** la **gestion** des **threads**. Ainsi, il permet d'avoir un **nombre fixe** de **threads**, facilement **identifiés**, et ainsi facilement **arrêtable**.

Il faut savoir qu'un avion n'a **aucune volonté propre**. Il ne peut agir que **lorsqu'on l'autorise**. Et cela semble logique. Il ne va pas se poser si des avions utilisent les pistes, il risque un crash.

Quand le thread **démarre**, il **crée** un **avion**. Ensuite, il **utilise** la **file de messag**e afin de communiquer au **contrôleur** qu'il est **prêt**.

La seule chose qu'il fait c'est attendre l'**autorisation** de la part du **contrôleur** pour agir. **Quand** on l'**autorise**, il décolle/atterrit. Enfin, s'il atterrit, il va se **garer** sur le **tarmac** et prend la première place disponible.

Enfin il change sa direction et recommence à attendre l'autorisation d'agir.

Le thread du contrôleur

Voilà le cœur du projet. Le contrôleur est le **SEUL** à **décider qui décolle** et **qui atterrit**. Afin d'y arriver, il doit préalablement **connaître tous les avions**. Ensuite, il établit une **liste d'attente qu'il met à jour à chaque action d'avion quelconque**. Il **autorise le premier** de chaque liste pour chaque piste et **met en attente** les autres avions.

Dans ce cas là, comment traitons-nous les atterrissages d'urgence ? **Il n'y a pas vraiment d'urgence.** Juste des mises à jour de listes. Et ces mises à jour de pistes sont **déclenchées par la libération d'une piste**. **Tout est liste d'attente** et c'est ce que je vais expliquer.

La liste d'attente

Il va y avoir **deux algorithmes** afin de faire fonctionner la liste d'attente. **Avant** le début du fonctionnement et **pendant**. Le but ici est de **créer** notre liste d'attente et ensuite de la **mettre à jour** lors de la libération d'une piste.

Première génération

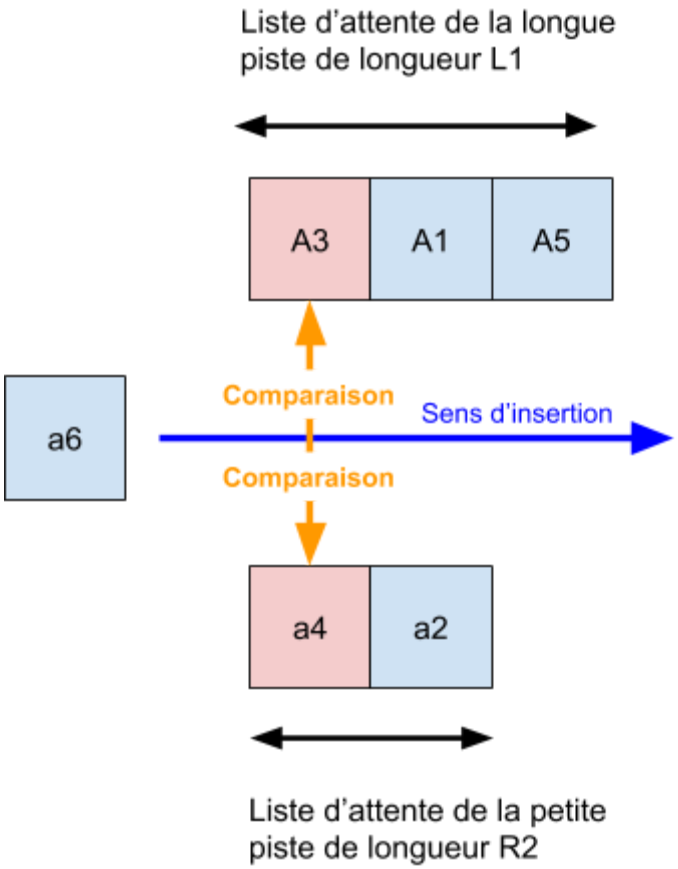
Lors de la prise de connaissance de nos avions, le contrôleur "va" **créer** non pas une mais **deux listes d'attente** de tous les avions pour chaque piste, en respectant l'ordre de priorité :

1. Piste **compatible** : on se demande sur quelle piste l'avion peut agir
2. Atterrissages **urgents** : on met d'abord les atterrissages **forcés** ou ceux en **manque de fuel**
3. **Date** le plus tôt : on fait arriver et partir les avions le plus tôt selon leur date prévue
4. **Longueur** de la liste d'attente : finalement, on tente d'équilibrer les longueurs de listes d'attente

Vous vous demandez sûrement où est la prise en compte du nombre de places sur le tarmac ? C'est très simple, ici il n'y en a pas. Au démarrage, tous les avions font des

atterrissages, ce qui “devrait” remplir le tarmac. Cependant, mes choix font que dès qu’un avion a atterri, il est directement reprogrammé pour un départ et remis dans la liste d’attente.

La difficulté ici, c’est que nous devons prendre en compte les deux listes d’attente. J’ai donc décidé de réaliser ce que j’appelle une “**Insertion par comparaison parallèle**”. Un schéma pourra vous aider à comprendre :



Mise à jour

Pendant le fonctionnement de l’aéroport, on **met à jour** les listes après l’utilisation d’une des pistes. Il y a 3 étapes :

1. **Toutes** les files sont **mises à jour** avec les attentes des pilotes. Ce rééquilibrage **nécessite toute les files** car l’attente totale doit être optimisé. Ce rééquilibrage ressemble à un un décalage/changement de piste.
2. Les **2 “nouveaux” avions** maximum sont donc **insérés** comme introduit précédemment. L’insertion peut se faire **après** car les listes sont **déjà triées**.

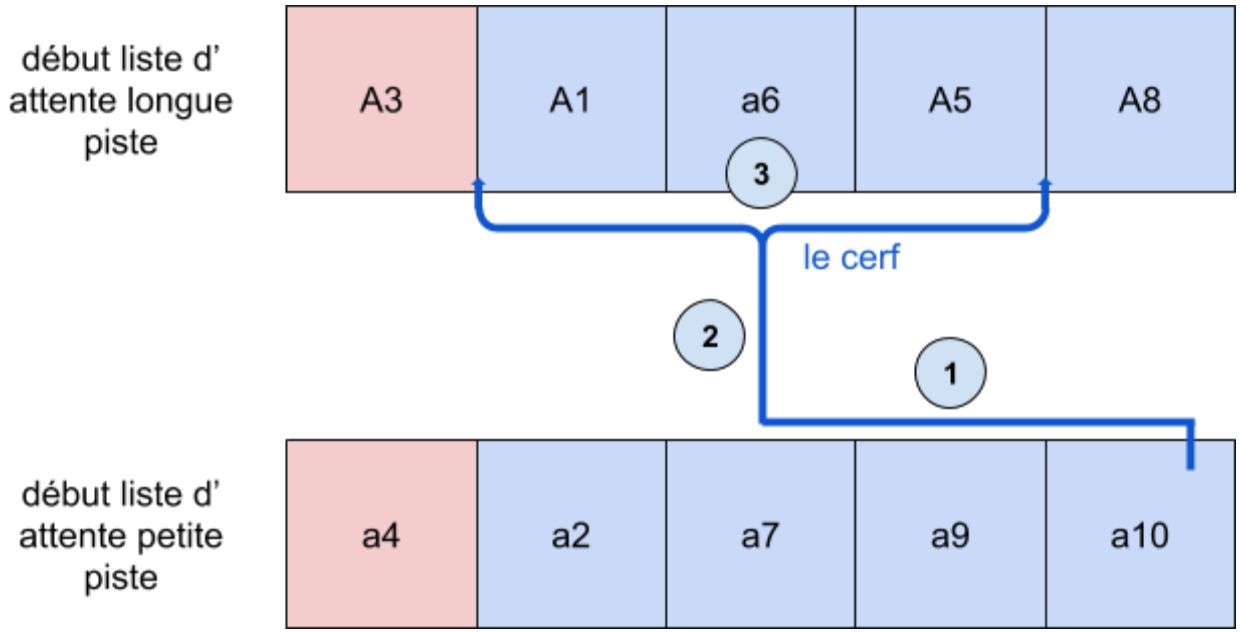
3. Le **premier** avion de chaque liste est autorisé.

Comparer les avions

Afin d'insérer et replacer les avions dans la file, on va devoir "**comparer**" les avions. Ainsi, on crée une fonction prenant comme **argument deux pointeurs d'avions**. La fonction elle retourne **-1 0 ou 1** si le premier avion est plus urgent que le second. On utilisera les critères cités précédemment.

Détail de la mise à jour

On commence par la **plus grande piste** car on a moins de chance de rabattre des avions de la grande piste vers la petite. On commence aussi par le **début**, car, la liste étant déjà triée la fois précédente, on aura **moins de déplacements** le long des listes d'attentes. Je surnomme cet algorithme, l'algorithme du "**cerf**":



1. On essaie de **s'avancer le plus possible** dans sa liste d'attente en fonction des **priorités** (avec en plus place sur le tarmac)
2. Si on est **compatible** avec l'autre liste d'attente et qu'on peut **équilibrer** le temps d'attente, on passe sur l'autre liste
3. On se **repositionne** sur l'autre liste en fonction des priorités

Ce programme peut paraître simple en théorie, leur implémentation n'en est pas moins fastidieux.

Configuration du programme

Le fichier *constantes.h* à la racine du projet permet de **contrôler** le **modèle** et ses **paramètres**.

On trouve beaucoup de paramètres **ajustables**, c'est pour ça qu'on :

- dès la création des avions :
    - Le nombre maximum d'avions
    - Les quantités min et max de fuel
    - La probabilité de vol international
    - La probabilité d'avoir un gros avion (utilisant la grande piste)
    - La probabilité d'un atterrissage forcé ou encore la capacité du tarmac.
- Durant la gestion de la piste :
    - Le coût en fuel de la mise en attente
    - La durée du décollage/atterrissage.

Vous pouvez les **modifier** afin d'avoir un **comportement particulier** et voir l'aéroport se comporte.

Limites du projet et questionnements

La limite du projet et de son comportement porte dans les paramètres configurables. **Il ne peut pas y avoir de crash d'avions**. Par là, je dis que **tous les avions** finissent toujours par **atterrir** et décoller.

Mon modèle a beau être **parfaitement logique et priorisé** , **je ne peux pas y faire grand chose** : Il faudrait faire des calculs précis sur les temps de décollage/temps sur le tarmac/brûlage de carburant...

La gestion d'obstacle n'aurait pas été trop un problème mais les heures des pilotes auraient été un **problème** de plus afin d'**équilibrer le modèle**. Il faudrait **fausser** les **paramètres** avant d'avoir un **modèle sûr** et **opérationnel** avec un **taux de russite de 100%**.

Je me suis demandé si je devais **autoriser l'utilisation de la piste** dans l'**attente** de **place** sur le **tarmac**. La question paraît simple au premier abord mais pourrait mener au

blocage du programme. Si notre tarmac est plein et qu'on fait **atterrir** deux avions, il nous sera impossible d'en faire **décoller un**, car les **pistes** sont **occupées**. Je prends déjà compte de la place sur le tarmac mais je me demandais si je pouvais **optimiser** aussi le **stationnement** sur le **tarmac**. Ce n'est pas un point essentiel donc j'ai décidé de me concentrer sur les pistes.

Merci

Merci d'avoir eu la patience de lire tout ce que j'ai écris.