# Rapport Projet LO41

**Université de technologie** de Belfort-Montbéliard

# Sujet Aéroport

**FARDEAU-SPIRE Malek**

**Semestre P20**

utbm
université de technologie
Belfort-Montbéliard

## Tables des matières:

## 1) Rappel du Sujet :

Le but du sujet était d'implémenter un aéroport et de gérer toutes les notions de contrôle en découlant, le tout en C une utilisant les divers objets et notions étudiées durant le semestre de LO41. L'aéroport devait permettre de faire circuler un grand nombre d'avions simultanément, à la taille et aux destinations différentes, et d'assurer la gestion des décollages/atterrissages des ces derniers sur 2 pistes différentes. Cela impliquait l'implémentation d'une tour de contrôle, permettant de communiquer avec les avions pour leur envoyer les informations de vol, les orienter vers les pistes disponibles correspondant à leur gabarit, ainsi que d'ordonner leur décollage pour que chacun puisse partir à l'heure tout en empêchant le risque de collision entre 2 entités sur la piste. Le sujet demandait aussi d'implémenter des cas de dysfonctionnement sur les pistes, ainsi que des avions au comportement différent face aux information de la tour de contrôle via l'implémentation d'un mode de fonctionnement propre à chaque avion.

## 2) Choix d'implémentation :

Pour répondre à ces problématiques, il a tout d'abord fallu réfléchir à la manière dont ces contraintes se traduisaient en langage informatique, et comment les implémenter, en utilisant notamment les concepts liés aux systèmes d'exploitation et aux objets bas niveau vus durant le semestre.
La première contrainte qui vient se poser est donc de pouvoir implémenter une tour de contrôle ainsi que des avions fonctionnant **simultanéments** et indépendaments les uns des autres. Pour cela, nous avons vu durant le semestre comment créer des processus ou des threads, qui permettent de paralléliser le travail du processeur et donc d'effectuer des actions de manière non linéaire comme dans un programme classique.
Le choix que j'ai fait à donc été de créer un **processus** fils en forkant au début du programme afin de créer le processus central de la tour de contrôle, tandis que le processus père lui continue de boucler pour créer autant d'avions que souhaités, implémentés via des **threads**, puis enfin de s'occuper de la bonne fermeture de tous les éléments et objets IPC utilisés lors du programme avant qu'il ne se termine.

La question qui vient ensuite est : comment est-ce-que ces programmes vont fonctionner, s'organiser et communiquer entre eux ? Je voulais que l'avion à sa création connaisse ses informations de vol basiques comme sa destination ou son gabarit, puis les envoie à la tour de contrôle qui va ensuite pouvoire répondre en

donnant des informations de vol plus complètes via les connaissances en sa possession, comme le choix de la piste ainsi que l'heure de décollage/atterrissage sélectionnée. Pour répondre à cette problématique, j'ai donc choisi d'implémenter des **files de messages**, utilisées réciproquement par chaque thread puis par la tour de contrôle, le tout via des structures contenant toutes les informations de vol nécessaires.

Ensuite, pour garantir un accès aux pistes sans **collisions** et pour éviter tout problème de synchronisation entre les différents acteurs du système, il est apparu évident que l'implémentation de **sémaphores** était obligatoire, pour pouvoir mettre en attente puis déverrouiller les différents éléments du programme. Cependant, les sémaphores seules ne suffisent pas, elles sont souvent associées à des variables monitrices, permettant par exemple dans notre cas à savoir le nombre d'avions en attente pour un piste donnée, le nombre d'avions en attente d'information, ou encore si une piste est utilisée ou pas, pour s'assurer de ne pas bloquer par sémaphore des éléments qui ne devraient pas l'être. J'ai donc choisi pour cela d'utiliser un **segment de mémoire partagée**, commun aux threads mais aussi au processus, via une structure contenant toutes les variables monitrices qui me sont utiles. Enfin, pour éviter un accès et une modification de ces variables critiques en simultané par 2 éléments du programme, pouvant causer des problèmes de valeurs inattendues, j'ai donc utilisé des **sémaphores mutex** autour de toutes les variables partagées afin de garantir l'accès et la modification de ces dernières.

Enfin, la réécriture des **signaux** est mise en place pour permettre de pouvoir fermer le programme proprement via les raccourcis clavier habituels sans avoir de pertes de mémoires ni d'objets IPC résiduels dans le système. De plus, pour faciliter le travail sur l'écriture du programme, j'ai dû réaliser un **makefile** afin d'automatiser le processus de compilation, qui se fait qui plus est ici avec des options de compilation spécifiques comme -pthread par exemple.

## 3) Fonctionnement et explication du programme :

Après avoir expliqué les éléments systèmes composant le squelette de l'aéroport et la raison du choix de leur utilisation, il est temps de passer au fonctionnement du programme en lui même. Tout d'abord, avant même l'exécution, l'utilisateur est invité à saisir le nombre d'avions maximum qu'il souhaite faire parcourir dans la simulation en argument du programme.

```
mfardeau@mfardeau-Predator-PH517-51:~/Bureau/LO41/Projet/AeroportLO41$ ./main
Erreur : Veuillez passer en argument le nombre total d'avions pour cette simulat
ions: Success
```

Ensuite, avant même de commencer à créer les threads, tous les objets IPC sont créés, les variables du segment de mémoire partagée sont initialisées, et les signaux permettant d'assurer la bonne fermeture du programme sont reliés à leur traitants respectif.

```c
void traitantSIGQUIT(int num)
{
    if (num!=SIGQUIT)
    {
        perror("Erreur signal SigQuit");
    }
    else
    {
        deletesem();
        deleteshm();
        deletemsgfile();
        kill(pid, SIGKILL);
        exit(0);
    }
}
```

```c
void traitantSIGINT(int num)
{
    if (num!=SIGINT)
    {
        perror("Erreur signal SigInt");
    }
    else
    {
        SharedMemory->exitrequested = 1;
    }
}
```

En effet, là où le signal SigQuit ( Ctrl - \ ) permet de fermer le programme instantanément, le signal SigInt ( Ctrl - C ) lui change uniquement la valeur d'un booléen exitrequested. Cela entraîne alors l'arrêt de la boucle dans le main faisant apparaître de nouveaux threads/avions, mais permet que le programme se termine normalement pour ensuite s'assurer de bien supprimer tous les threads, processus et objets IPC proprement.

Enfin une fois tous les éléments initialisés, le processus de la tour de contrôle est créée, et la boucle de lancement des threads est lancée. Le fonctionnement de l'aéroport peut alors commencer.

Quand un avion est créé, il génère en premier lieu ses information de base dans la structure PlaneInformation puis les envoie à la tour de contrôle via la file de message avant d'attendre ses informations de vol en retour.

```
Avion n°714 : En attente d'informations : Gros Porteur en provenance de Madrid/B
arajas - Adolfo-Suárez
```

Ces information initiales contiennent le gabarit de l'avion, sa direction (atterrissage / décollage), son numéro d'immatriculation ou encore son niveau de carburant initial. Ces informations sont donc envoyées à la tour de contrôle, qui va alors les afficher puis générer à partir des ces dernières les informations de vol complètes nécessaire pour l'avion.

```
Avion n°714 : Informations de vol : Piste 1 : Arrivée prévue à 1:46, délai max 2
 min, avion en mode Assuré
```

Tout d'abord, le choix de la piste est effectué. La piste 1 étant la seule pouvant accueillir les avions gros calibre, ces derniers sont automatiquement envoyés sur cette dernière. Cependant, les avions de petit calibre peuvent être envoyés sur la grande piste lorsque la piste plus petite est surchargée.

```
Tour de contrôle : Piste 2 : Surcharge, avion petit calibre envoyé vers piste 1
```

Ensuite, des strings donnant la route et le chemin à suivre sont ajoutés.
Enfin, le mode d'opération de l'avion est choisi en fonction de son niveau de carburant initial, entre mode Normal, Assuré et Urgent. Ce mode va ensuite permettre de déterminer l'heure de décollage, de instantanément jusqu'à 2 minutes après l'heure actuelle via l'utilisation de la struct tm de la bibliothèque <time.h>, ainsi que le délai maximum autorisé, compris entre 1 et 3 minutes.

Une fois que l'avion a reçu ses informations en retour, il va ensuite attendre son heure de décollage. Pour cela, il boucle jusqu'à l'atteindre et perd une unité de carburant dans le même temps à chaque itération. Cette gestion du carburant peut donc amener un avion fraîchement créé à passer de mode Normal à mode Assuré, puis de mode Assuré à mode Urgent. Lorsqu'un avion passe en mode Urgent, cela signifie que ses réserves de carburant sont arrivées au plus bas, et il va donc essayer d'atterrir le plus vite possible sans devoir attendre son heure de décollage initiale.

De plus, des obstacles générés par la tour de contrôle peuvent apparaître de temps à autres sur les pistes et bloquer ces dernières pendant un temps donné. C'est donc là que le mode de l'Avion entre de nouveau en jeu, car à la création comme au changement d'état, un avion qui passe en mode Assuré ou en mode Urgent vas envoyer un signal (SIGUSR1 / SIGUSR2) à la tour de contrôle. Un avion en mode

assuré va appeler une équipe pour enlever plus rapidement l'obstacle gênant sa piste, tandis qu'un avion en mode Urgent appellera une équipe de techniciens professionnels permettant de libérer quasi instantanément sa piste assignée de tout obstacle pouvant se trouver dans le passage.

```
Tour de contrôle : Piste 1 : Obstacle inconnu
```

```
Avion n°717 : En attente piste 1 : Passage en Mode Urgent
Tour de contrôle : Piste 1 : Envoi équipe pour enlever obstacle instantanément
Tour de contrôle : Piste 1 : Obstacle enlevé
Avion n°717 : Piste 1 : décollage
```

Une fois que l'heure pour l'avion d'atterrir est arrivée ou alors que ses réserves de carburant sont trop basses, il va se mettre en attente sur sa piste avant d'atterrir / décoller ce qui va la laisser utilisée pendant un certain temps. Toutes ces opérations d'attente de message, de décollage, ainsi que tous les accès aux variables partagées sont bien sûr sécurisés par les sémaphores tout le long du programme. Enfin, comme toute dernière action, une fois que l'avion aura libéré sa piste, il enverra un message d'alerte à la tour de contrôle si son délai max à été dépassé, avant que finalement le thread ne se ferme de lui même.

```
Avion n°718 : Alerte : Le délai max a été dépassé de 1 min
```

En supplément, toutes les variables permettant de réaliser les choix dans le programme, comme le délai avant la création d'un nouvel avion, le temps de décollage, le niveau d'essence initial ou encore la fréquence d'apparition des obstacles sont toutes regroupées dans un fichier constants.h afin d'être facilement accessibles et modifiables pour pouvoir tester la simulation dans tout type de configuration.

## 4) Difficultées rencontrées :

-Etant donné que les structures des informations de vol contiennent des strings, et que l'implémentation des strings est assez limitée en C comparée au C++ (tableaux de caractères), il m'était impossible d'utiliser ces structures dans des fonctions et de faire des retours par valeur car cela ne fonctionnait pas pour les chaînes de caractères. J'ai donc dû initialiser ces structures d'information en variable globale dans les threads et dans le processus contrôle pour pouvoir les utiliser ensuite facilement dans toutes les fonctions membres. Le problème qui s'est présenté était que comme la structure était déclarée en variable globale, elle était alors la même variable partagée par tous les threads au lieu d'être indépendante à chacun. J'ai

donc appris à utiliser le mot clé __thread, qui permet de déclarer une variable globale unique à un thread.

```c
#include "randompart.h"

//Mot clé __thread nécessaire pour créer une variable globale unique à
__thread FlightInformationStruct LocalFlightInformation;
__thread PlaneInformationStruct LocalPlaneInformation;

void * plane (void * arg)
{
```

- Problème rencontré avec les signaux : Lors de l'envoi du signal Ctrl-C, que j'ai réecrit pour simplement arrêter l'apparition de nouveaux avions mais pour laisser le programme se finir normalement, le programme se bloquait sans raison apparente et il ne pouvait pas se terminer. J'ai alors appris que les signaux, étant des objets extrêmement bas niveau, "entraient en collision" avec les files de messages, ce qui empêchait les informations de vol d'être envoyées et donc le programme de bien se terminer. J'ai donc dû ajouter pour une exception pour que les messages dont l'envoi a échoué pour cause de croisement avec des signaux soient relancés automatiquement, permettant au programme d'achever sa fermeture.

```c
void sendPlaneInformation()
{
    errno = 0;
    if (msgsnd(msg_id, &LocalPlaneInformation, sizeof(PlaneInformationStruct) - sizeof(long),0) == -1)
        {
            //En cas de Ctrl-C tout message envoyé est interrompu, on le relance afin que le programme se ferme correctement
            if (errno == EINTR)
                sendPlaneInformation();
            else
            {
            perror("Erreur d'envoi message avion");
            //exit(EXIT_FAILURE);
            }
        }
}
```

- J'ai aussi rencontré de nombreux problèmes durant le développement du programme, et pour les résoudre j'ai appris à utiliser gdb (en créant une option make debug dans mon makefile) afin de mieux comprendre l'origine des problèmes rencontrés. Cela s'est avéré très pratique, notamment pour afficher clairement quand un thread ou un processus est créé, supprimé, ou qu'il rencontre un problème.

```
    $(CC) $< -o $@ $(CFlags)

debug :
    gdb -ex=r --args main 10
```

```
[Detaching after fork from child process 20499]
[New Thread 0x7ffff77c4700 (LWP 20500)]
Avion n°714 : En attente d'informations : Petit Calibre à destination de Madrid/
Barajas - Adolfo-Suárez
```

## 5) Améliorations possibles :

-Pour l'utilisation de la mémoire partagée, j'ai simplement déclaré et utilisé une structure SharedMemory contenant toutes les variables qui m'étaient nécessaires. Après discussion avec mon enseignant de TP, il m'a dit qu'il était recommandé de créer un segment de mémoire partagée par variable et j'aurais donc pu améliorer ce point.

-J'ai choisi au début de la réalisation de ce projet de réaliser la tour de contrôle via un processus fils car son rôle me paraissait différent et plus important que celui des avions mais aussi pour montrer et utiliser ce que j'avais appris. En y réfléchissant après coup, je pense que rien ne m'aurait empêché de tout faire en thread et cela m'aurait épargné en plus la création du segment de mémoire partagée. J'aurais sans doute amélioré ce point dans une optique de simplifier le programme en dehors du contexte d'un projet à rendre dont le but est aussi de montrer mes compétences acquises en LO41.