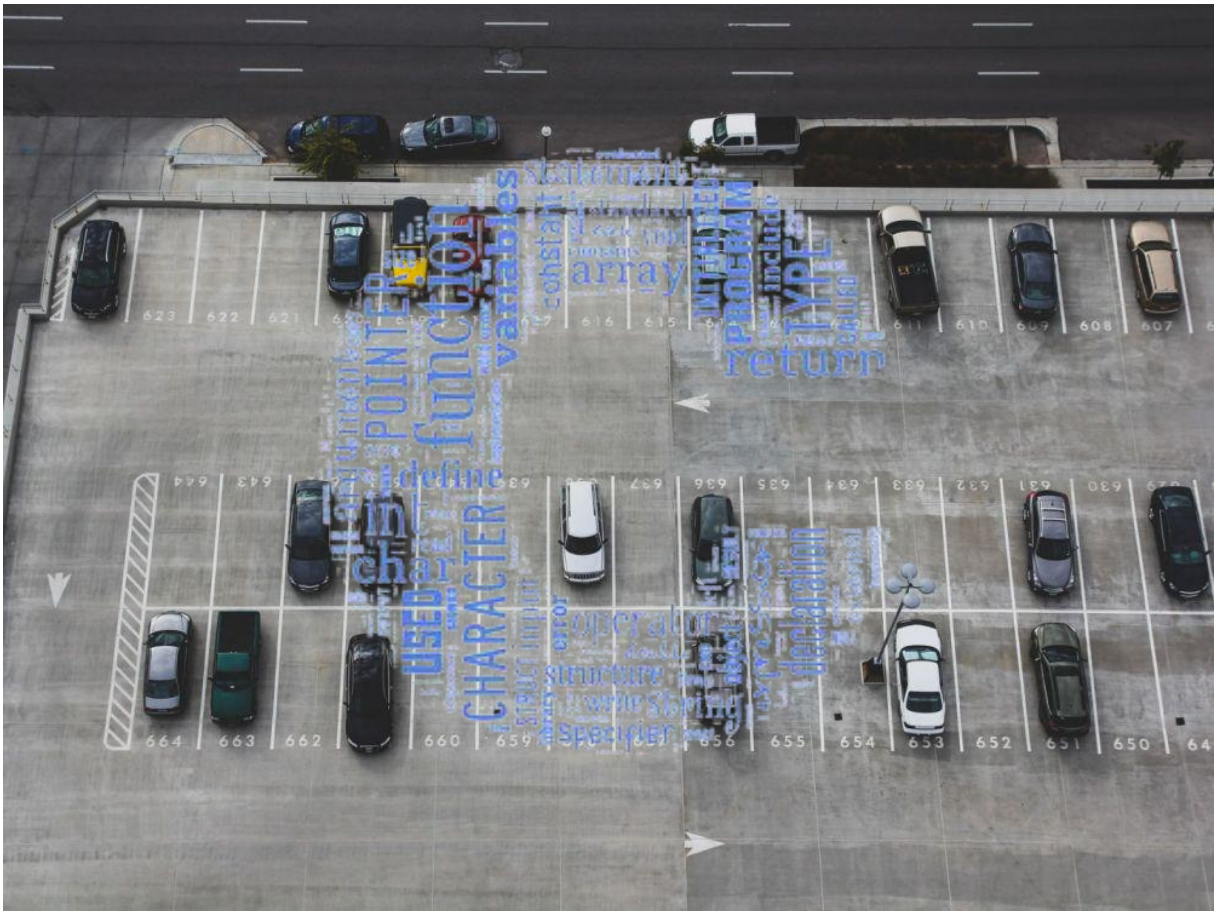
utbm
université de technologie
Belfort-Montbéliard

Semestre printemps 2022

Rapport de projet - SY40



PROJET LANGAGE C
SIMULATION D'UN PARKING

Projet réalisé par

Jérémy KIEFFER

Titouan BURGY

Projet encadré par

Philippe DESCAMPS

Sommaire

I. Présentation du projet

1. Description du projet

Dans le cadre de l'UV SY40 (Architecture des systèmes d'exploitation) en première année d'Informatique à l'UTBM, il nous a été demandé de réaliser un projet pour mettre en pratique les différentes connaissances acquis durant le semestre avec comme finalité la mise en place d'une simulation d'un parking public à l'aide du langage de programmation C. Les différentes consignes pour ce projet sont affichées ci-dessous :

Les parkings publics ou parcs de stationnement sont dorénavant considérés comme des zones « tampon » permettant d'offrir des solutions efficaces et fiables de mobilité urbaine. Que ce soit en périphérie des villes (P+R) ou au centre de celles-ci, ils contribuent notamment à la gestion du transport de proximité et au désengorgement urbain.

Le projet s'intéresse à la gestion de parking sécurisé. Deux types d'usagers sont autorisés : les abonnés et les non abonnés.

- Les abonnés du parking possèdent un émetteur de signal qui sert de code pour pouvoir entrer dans le parking. Si le système reconnait le signal que l'utilisateur vient d'envoyer, la barrière s'ouvre automatiquement. C'est un système d'automation et de gestion des aires de stationnement de parkings 24h/24 et 7j/7. Le contrôle d'accès est réalisé automatiquement par l'authentification des utilisateurs et la durée de l'abonnement.
- Les non abonnés ont accès à une borne de distribution de ticket. Ces tickets sont proposés à condition qu'une aire de stationnement est disponible.

On considère que le parking dispose de N places de parking privés accessibles aux abonnés et M places disponibles pour les non abonnés. Dans les faits, si un abonné ne peut pas accéder à une place réservée alors il est en droit d'occuper une place non réservée. Afin de faciliter la gestion du parking de nuit, nous considérons qu'à partir de 18h une zone de débordement est prévue. Cette zone couvre exclusivement les places non-abonnés. La contrainte sur cette zone se relâche d'heure en heure évitant la saturation du parking au plus tôt. L'entrée et la sortie se fait par la même voie. Celle-ci est gérer par un feu bicolore permettant d'informer l'utilisateur de la possibilité d'accéder à la barrière. Il ne peut y avoir simultanément deux véhicules à la barrière.

Livrable final à faire parvenir à : philippe.descamps@utbm.fr

- Rapport de projet comportant une explication sur vos choix de conception
- Code en zip (Makefile + Programmes + Fichier *Lisezmoi*)

2. Analyse des demandes

Après une première analyse des différentes demandes du projet, nous avons pu mettre en place une liste de problèmes à résoudre pour mener à bien ce projet :

- Quelle(s) méthode(s) utiliser pour la simulation des usagers, du parking etc … ?
- Quelles sont les différentes structures à mettre en place ?
- Comment gérer la création de processus non-stop ? Comment gérer le temps (primordial pour la mise en place de la zone de débordement) ?
- Quel affichage ? Intérieur du parking ou activités à la barrière ?

II. Les méthodes utilisées

1. Choix de conception – Les Threads

Notre choix pour la simulation des processus s'est assez vite tourné vers les threads. En effet, nous avons privilégié cette méthode pour plusieurs raisons :

- Les différentes fonctions vues en cours nous paraissaient adaptée aux différents problèmes que présente ce projet. Le thread étant un processus léger, la communication entre threads est beaucoup moins lourde et coûteuse que la communication entre processus système (dit lourds, comme l'utilisation de fork). De plus, étant donné que les threads se lancent en même temps que le processus main, la multiplication des threads et la synchronisation des différentes tâches pour la simulation du parking est largement simplifiée.
- D'un point de vue personnel, nous avons tous les deux bien compris et assimilé la notion de thread. Lors des différents travaux pratiques de l'UV, nous étions à l'aise avec l'utilisation des threads là où certains aspects d'autres méthodes nous paraissaient beaucoup plus compliqués pour effectuer la même action.
- La protection des variables ou des parties de code dont la synchronisation est très importante (comme le nombre d'usager dans le parking ou le nombre d'usager qui attende à la barrière) pouvait très simplement être résolue à l'aide des mutex :

```
/* Passage de la barriere*/
countB++; // accède à la barrière
printAction("Accede à la barriere", u_p->id, 0, u_p->isAbonne);
pthread_mutex_unlock(&mutex);
sleep(1); //temps pour passer la barriere
pthread_mutex_lock(&mutex);
countB--; // s'eloigne la barriere
printAction("Entre dans le parking", u_p->id, 0, u_p->isAbonne);
```

Dans l'exemple ci-dessus, on remarque en effet que le principe de « passer la barrière » est très important. Dans la consigne il est en effet précisé qu'un seul utilisateur peut passer de l'autre côté et le mutex nous permis de respecter ce principe-là en rajoutant simplement deux fonctions (implémenter avec la librairie des threads).

2. Mise en place des structures

Pour le bon fonctionnement du projet, nous avons mis en place trois structures particulières dont nous allons vous expliquer ci-dessous les détails.

a. Abonné et non-abonné

```
typedef struct Usager
{
        int id; //numero d'identification de l'usager
        bool isAbonne; //true si abonne
        int stationnement; //numero de la place (-1 si non stationné)
}Usager;
```

La première structure mise en place est celle qui permet de représenter les utilisateurs de notre parking. Comme vous pouvez le voir, nous l'avons nommé « Usager » et cette dernière possède trois champs différents. Le premier (id) est simplement l'identifiant de l'usager. Dans notre cas, ce dernier est en fait l'ordre de création des threads (expliqué un peu plus bas), c'est-à-dire que le premier usager créé aura l'id 1, le deuxième aura l'id 2 et ainsi de suite … Le deuxième champ est un booléen (isAbonne) qui, comme cela est noté en commentaire du code, est à vrai si l'usager est un abonné et à faux s'il ne l'est pas. Le dernier champ est sans doute le plus important : c'est un entier (stationnement) qui contient le numéro de la place à laquelle est garé l'usager (s'il n'est pas garé, cet entier est mis à -1). Grâce à cet entier, nous pouvons mettre à jour l'affichage sur le terminal (voir Affichage sur le terminal) et gérer finalement les différentes actions que peuvent réalisées l'usager (thread).

b. Parking

```
typedef struct PlaceParking
{
        int id; //numero de la place
        bool isAbonne; //true si place réservé aux abonnés
        int idUsager; //id de l'usager sur la place (-1 si aucun)
}PlaceParking;
```

La deuxième structure que nous avons créée est celle permettant de représenter le parking en lui-même. Sur l'image ci-dessus, vous pouvez voir que nous avons appelé la structure « PlaceParking » qui représente comme son nom l'indique une place de parking. Elle possède trois champs : l'entier id (numéro de la place), le booléen isAbonne (même chose que pour la structure « Usager », est à vrai si la place est réservée à un abonné, faux si elle ne l'est pas) et l'entier idUsager qui va contenir l'id de l'usager qui est actuellement garé sur cette place). Enfaite, le parking va être lui représenté par un tableau de « PlaceParking » :

`PlaceParking parking[NUM_P];` NUM_P étant une constante définie dans le fichier « CodingFunctions.h » comme NUM_P_ABONNE (nombres places pour abonné) + NUM_P_NABONNE (nombres places pour non-abonné) dont les valeurs peuvent être modifiées pour tester le programme avec différentes tailles de parking et même différentes proportions de place réservée aux abonnés.

c. Chronomètre

```
typedef struct Heure
{
        int min; //les minutes
        int h; //les heures
}Heure;
```

Pour pouvoir mettre en place le système de débordement à partir de 18h, il nous fallait une structure pour gérer le parking en fonction de l'heure. Nous avons donc ajouté au programme la structure « Heure » qui possède deux champs : un entier (h) pour gérer donc l'heure fictive et un deuxième entier (min) qui représente les minutes. Ce deuxième entier a été implémenter pour plus de compréhension sur la notion du temps dans notre simulation.

III. Implémentation des méthodes

1. Génération des threads - usagers

Lors de nos premiers tests, nous implémentions les threads via la création à un nombre limité de ces derniers (c'est-à-dire que nous créions un nombre précis de thread). Cependant, une question a très vite été ramener sur le devant de la scène : Comment faire pour que la génération des threads se passent en illimités ? Nous avons donc décider d'implémenter une boucle while qui créerait et détruirait sans cesse de nouveaux threads :

```
while(true){
        create_threads();
        end_threads();
}
```

```
//Fonction pour générer les threads usager
void create_threads()
{
        long tab[2]={0};
        while(true){
                tab[0]=numThread;
                attente_aleatoire(3);
                if(boolean_aleatoire()){
                        tab[1]=1;
                        if(pthread_create(tidUsager+numThread,0,(void *(*)())fonc_usager, (void *) tab) == -1)
                        {
                                debug("Creation thread abonne");
                        }
                }else{
                        tab[1]=0;
                        if(pthread_create(tidUsager+numThread,0,(void *(*)())fonc_usager, (void *) tab) == -1)
                        {
                                debug("Creation thread non-abonne");
                        }
                }
                numThread++;
        }
}
```

```
//Fonction pour terminer les threads usager
void end_threads()
{
        for(int i=0;i<numThread;++i)
                pthread_join(tidUsager[i],NULL);
}
```

De cette manière, nous pouvions créer des threads « à l'infini » avec une attente aléatoire (cette attente peut être comprise entre 1 et 3 secondes pour éviter la création d'une multitude de threads en très peu de temps, ce qui provoque des bugs). De plus, à l'aide de la fonction boolean_aleatoire (qui comme son nom l'indique renvoie au hasard true ou false), nous avons pu générer une proportion aléatoire d'abonné et de non-abonné.

2. Comportement des usagers

- Usager Extérieur

Cette fonction porte sur l'ensemble des actions à réaliser lorsque l'usager se trouve à l'extérieur du parking et souhaite accéder à la barrière pour entrer dans celui-ci. Chronologiquement, l'usager vérifie dans un premier temps, si la barrière est libre et se met en attente dans le cas contraire. Une fois son tour venu, il accède à la barrière et entre dans le parking, puis, il avertit l'usager suivant (en priorité les usagers attendant à l'intérieur du parking) s'il y en a un. Finalement, il cherche sa place et stationne.

(countB est la variable concernant le nombre de personne à la barrière)

```
--------------- Usager Exterieur ---------------

Lock(mutex)
Si (countB > 1) Alors
    nbAttExt++
    Unlock(mutex)
    Ext.wait
    Lock(mutex)
    nbAttExt--
Fsi

countB++ //accède barrière
Unlock(mutex)
attendre(2) //simule le temps passé à la barrière
Lock(mutex)
countB-- //s'éloigne de la barrière
ChercheStationnement() //cherche un stationnement

Si (nbAttInt > 0) Alors
    Unlock(mutex)
    Int.signal
Sinon
    Si (nbAttExt > 0) Alors
        Unlock(mutex)
        Ext.signal
    Sinon
        Unlock(mutex)
    Fsi
Fsi
```

- Usager Intérieur

Cette fonction est assez similaire et intervient juste après la fonction précédente pour permettre à l'usager stationnant dans le parking, de sortir de celui-ci. En effet, le raisonnement est le même avec de légères différences. Premièrement, l'usager quitte sa place et vérifie, si la barrière est libre. Dans le cas contraire, il se met en attente. Lorsque son tour arrive, il accède à la barrière et sort du parking, puis, avertit l'usager suivant (en priorité les usagers attendant à l'intérieur du parking).

```
--------------- Usager Interieur ---------------

Circuler() //quitte le stationnement

Lock(mutex)
Si (countB > 1) Alors
    nbAttInt++
    Unlock(mutex)
    Ext.wait
    Lock(mutex)
    nbAttInt--
Fsi

countB++ //accède barrière
Unlock(mutex)
attendre(2) //simule le temps passé à la barrière
Lock(mutex)
countB-- //s'éloigne de la barrière

Si (nbAttInt > 0) Alors
    Unlock(mutex)
    Int.signal
Sinon
    Si (nbAttExt > 0) Alors
        Unlock(mutex)
        Ext.signal
    Sinon
        Unlock(mutex)
    Fsi
Fsi
```

- Fonction Usager

Cette fonction permet de gérer les 2 précédentes. En effet, dans le cas où le parking ne peut plus accueillir d'autres usagers, il faut leur interdire l'accès. C'est donc le rôle de cette fonction, qui initialise l'usager en question, et vérifie le nombre de place du parking disponible (en fonction des usagers qui sont déjà en train d'attendre ou qui sont à la barrière). Si les conditions ne sont pas respectées, l'usager s'en va. Dans le cas contraire, les autres fonctions sont lancées (UsagerExterieur, TempsDeStationnement, UsagerInterieur).

```
--------------- Fonction Usagers ---------------

InitialisationUsager() //initialiser un Usager

Lock(mutex)
condition <-- nbAttExt - countB - debordement

Si (condition <= 0) Alors
    Unlock(mutex)
    // Usager part car aucune place libre
Sinon
    Unlock(mutex)
    UsagerInterieur()
    TempsDeStationnement()
    UsagerExterieur()
Fsi
```

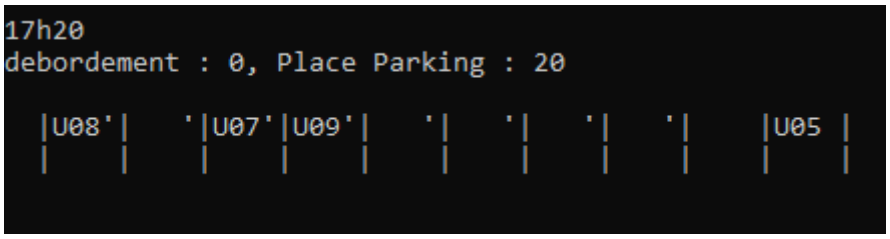
- Pour finir, toutes ces fonctions utilisent d'autres fonctions plus simples permettant de relier les usagers au parking comme :
    - nbPlaceLibreparking : renvoie le nombre de place libre du parking pour un usager en paramètre (abonné ou non-abonné).
    - stationner : permet de faire stationner un Usager sur une place de parking.
    - circuler : permet de faire sortir un usager d'une place de parking.
    - attente_aleatoire : met en attente un usager pendant une durée aléatoire.
    - chercheStationnement : cherche une place adaptée pour un usager.
    - calculDebordement : calcul le débordement en fonction de l'heure qu'il est.

3. Affichage sur le terminal

1. Partie gauche : Heure + places de parking

Cette partie est affichée chaque seconde grâce :

- À la fonction printParking qui permet d'afficher le parking.
- Au thread chrono (fonc_chrono) qui incrémente le chrono de 10 minutes chaque seconde pour simuler le temps qui passe.



On retrouve plusieurs données comme l'heure, le nombre de place de parking et le nombre de place retirer à cause du débordement. De plus, un affichage console a été mis en place de manière à simuler des places de parking abonné (| '|) ou non-abonné (| |). Si un numéro d'usager apparaît, ça signifie que la place est prise par cet usager.

2. Partie droite : Actions des usagers

Cette partie est affichée grâce à la fonction printAction qui permet, si elle est activée, d'afficher à droite de l'écran toutes les actions réalisées par les usagers. On retrouve comme différentes actions comme :

- Patiente
- Accède barrière
- Se gare sur une place
- Quitte sa place
- Entre dans le parking
- Sort du parking

Par exemple « U6 (A) : Quitte sa place » signifie que l'usager numéro 6, qui est abonné, quitte sa place de parking.

4. Conclusion

1. Nos impressions

Lors de ce projet, nous avons pu développer des aspects sur les threads que nous n'avions pas pu forcément déployer lors des différents travaux pratiques (génération illimitée de threads, compréhension de l'importance des mutex etc …), tout en revoyant tous les aspects (surtout ceux liés aux threads) déjà vu en cours. Nous avons l'impression de nous en être plutôt bien sorti dans la mise en place de ce projet et la bonne collaboration a permis de mener un bon travail dans un temps assez restreint.

2. Les points d'améliorations

- L'affichage reste très simple, il est difficile de faire un affichage esthétique sur la console. De plus, il arrive que l'affichage rencontre quelques problèmes de disposition, notamment au niveau du parking qui est excentré par moment.
- Le système mis en place pour le débordement à partir d'une certaine heure peut être améliorer (actuellement, et ça malgré le débordement, les abonnés peuvent continuer à se garer : ce principe nous a cependant paru cohérent avec la réalité et nous avons donc décider de le laisser comme cela).