# Gestion du Parking Automobile

# MULTITACHE en C/C++ sous Linux

## B3427

## Maria Cosmina ETEGAN

## Henri Hannetel

### Spécification et Conception

1. **Conception graphique**

Annexes avec les 3 phases (initialisation, moteur, destruction) de chaque tâche (*mere, simulation, entree, sortie*)

1. **Commentaires sur les modifications apportées à la conception initiale**

Comme l’architecture faite pendant la première séance de TP était incorrecte et il était impossible de poursuive nous l’avons refaite.

Les principales erreurs que nous avons faites et les améliorations apportées après :

* Le choix des tâches était incorrect. Nous avions 2 tâches (*Clavier* et *Simulation*) qui n’avaient pas des rôles indépendantes et la tâche mère était absente dans l’architecture initiale. En plus la tâche *clavier* mettrait en place et libérait l’écran et le clavier – actions qui doivent être réalisées par la tâche *mere* dans la phase d’initialisation.

**Solution :**

Le remplacement de ces 2 tâches par une tâche *mere*  et une tâche *simulation :*

la tâche *mere*

-à l’initialisation de l’application met en place les ressources, crée les tâches *entree* (tâche porte dans l’architecture initiale), *sortie, simulateur,* lance la tâche *Heure*

*-* à la fin de l’application (commande Q) s’occupe de la destruction de ces tâches et la libération (ou la destruction) de ces ressources.

la tâche simulateur

-qui va gérer les commandes reçues du clavier et (si c’est le cas) lance la tache correspondante (*entree* – pour une commande P ou A ou *sortie* pour une commande S). Dans le cas où l’utilisateur demande l’arrêt de l’application – commande Q – ce tâche envoie un signal SIGUSR2 à la tâche *mere* pour demander l’arrêt (à la réception de ce signal celle-là va envoyer aussi un signal SIGUSR2 aux tâches *entree* et *sortie* pour demander l’arrêt )

* Pour les actions que nous avons représentées nous n’avons pas précisé le moment quand elles vont s’exécuter.

**Solution :**

La nouvelle architecture présente toutes les actions avec le moment quand elles vont s’exécuter. Nous avons utilisé toutes les méthodes et fonctions fournies et bien d’autres actions en précisant à chaque fois qui fait quoi et quand.

* Les 2 tâches *voiturier* (entree et sortie) lancées par les fonctions GarerVoiture et SortirVoiture n’étaient pas représentées

**Solution :**

Nous avons représenté les 2 tâches. Elles sont lancées par la fonction GarerVoiture et SortirVoiture dans la tâche *entree* respectivement *sortie*. A la fin normale de ces 2 tâches elles renvoient un signal SIGCHLD à la tâche *entree* respectivement *sortie* ainsi on pourra lancer plusieurs tâches *voiturier* en même temps. La fin de l’application (caractère Q tapé au clavier) est gérée par l’utilisation d’un signal SIGUSR2 envoyé par le tâches *entree* et *sortie* pour demander aux *voituriers* de s’arrêter.

* La mémoire partagée n’était pas détaillée

**Solution :**

Nous avons représenté la mémoire partagée d’une façon plus claire et détaillé.

On a 3 zones de mémoire :

-nbPlaces : une qui est un entier non signé qui peut prendre des valeurs entre 0 et NB\_PLACES et qui représente le nombre de places disponible dans le parking

-etatParking : une zone qui contient un tableau de 8 éléments représentant chaque place du parking. Chaque élément contient 3 champs : un de type TypeUsager (valeur initiale – AUCUN ), un entier représentant le numéro de voiture (valeur initiale - 0), un entier représentant l’instant d’arrivé (valeur initiale - 0)

-requetes : une zone qui contient un tableau de 3 éléments représentant les requêtes. Chaque élément contiens 3 champs : un de type TypeBarriere (valeur initiale - AUCUNE), un de type TypeUsager (valeur initiale - AUCUN) et un entier représentant l’instant de la demande (valeur initiale - 0).

Toutes ces zones étant des ressources critiques (les tâches *entree et* *sortie* vont y accéder en même temps) elles doivent être protégées donc on va mettre en place un sémaphore général avec 3 sémaphores élémentaires (un pour chaque zone mémoire) qui vont être des mutex de gestion de conflits d’accès

* Nous avons oublié de préciser qu’il existe 3 tâches *entree*, une pour chaque barrière

**Autres problèmes importantes que nous n’avons pas traités dans l’architecture initiale :**

* Communication entre la tâche qui gère les commandes (*Simulateur*) et les tâches *entree* et *sortie*

Nous avons mis en place 4 canaux nommés dont les trames contiennent le type de voiture (TypeUsager)et le numéro de place demandé. (un canaux entre *sortie* et *simulateur* et un entre chaque tâche *entree* et *simulateur*)*.*

* Vol de place

Dans le cas du parking plein et requêtes déposés, si on a une voiture vient de sortir du parking (fin de la tâche *voiturier sortie*) et la mémoire partagé est mise à jour (donc on déclare qu’il y a un place libre) et une voiture demande d’être garé en ce moment elle peut être garée même s’il y avait des requêtes déposée avant (quand le parking était plein) – ce voiture va être garé dès qu’elle arrive à la barrière et ainsi elle va voler le place d’une autre voiture qui avait déposé une requête (donc qui attendait déjà).

Pour éviter ces problèmes on a mis en place 3 mutex entre la tâche *sortie* et chaque tâche *entree* pour qu’elles puissent communiquer directement sans passer par la mémoire partagée. Ainsi quand le parking est plein et il y des requêtes déposés et une voiture vient de sortir du parking (fin *voiturier*) la tâche sortie accède au mutex pour annoncer la tâche *entree* qu’il y a un place libre – la tache *entree* va garer une voiture qui a déposé une requête dans la barrière correspondante.

### Réalisation