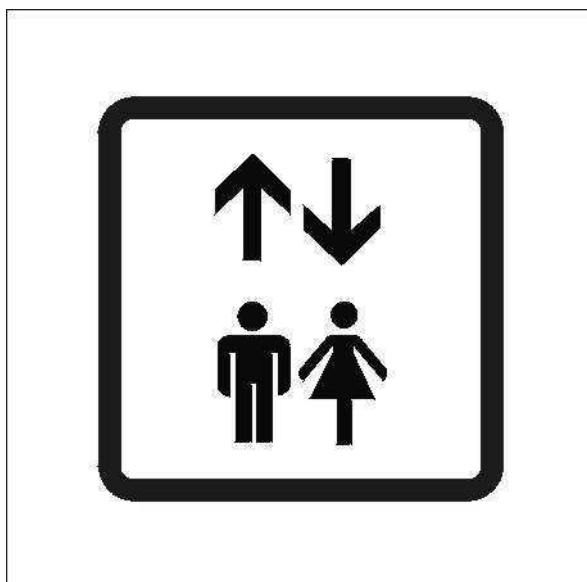


Rapport de projet : Ascenseur



SOMMAIRE

SOMMAIRE	2
Introduction.....	3
1. Architecture détaillée : modèle simple	4
1.1) Stockage et création des données.	4
1.2) Fonction relative à la file	5
1.3) Fonctions de traitement.....	7
1.3.1) Fonction déplacement	7
1.3.2) Fonction mode_pas_a_pas	7
1.3.3) Fonction continu	9
1.4) Fonction statistique.....	10
1.5) Interface homme/ machine.....	11
2) Architecture du projet : Modèle complexe	12
2.1) Types de fichiers utilisés.....	14
2.2) Définition des types utilisés	14
2.3) Architecture détaillée.....	15
2.3.1) Variables utiles du main.....	15
2.3.2) Création des données	16
2.3.2.1) Parseur	16
2.3.2.2) creation_requette_manuel.....	16
2.3.3) Structures et fonctions de stockage des données	17
2.3.3.1) init immeuble	17
2.3.3.2) chainage_immeuble.....	17
2.3.3.3) suppression_immeuble	17
2.3.3.4) elt_immeuble	18
2.3.3.5) init_ascenseur	18
2.3.3.6) chainage_ascenseur	18
2.3.4) Fonctions de traitement des données	19
2.3.4.1) Sélection et validation des requêtes en attente	19
2.3.4.2) requette_fini.....	20
2.3.4.3) Escalier	20
2.3.4.4) Déplacement	20
2.3.4.5) Stats	21
2.3.5) Affichage.....	21
2.3.5.1) afficahge_requette_attente.....	21
2.3.5.2) affichage_ascenseur	21
2.3.5.3) petit_cadre	21
3) Problèmes rencontrés // Méthode.....	22
Conclusion	23

Introduction

Le projet ascenseur nous a posé beaucoup de problème pour imaginer le fonctionnement de l'ascenseur. Nous avons beaucoup discuté et décidé de créer un modèle simple pour comprendre le fonctionnement et envisager des solutions pour un modèle plus complexe. Nous allons donc détailler dans le présent rapport le modèle simple avec les structures et les stockages de données, ainsi que les fonctions de traitement. Ensuite nous allons expliquer notre modèle complexe en détaillant les structures de stockage et de traitement des informations. Nous expliquerons aussi pourquoi ce modèle est plus performant que le modèle simple. Enfin nous terminerons par un récapitulatif de notre méthode de travail ainsi que des problèmes rencontrés lors du développement de ce projet.

1. Architecture détaillée : modèle simple

Dans un premier temps nous avons jugé judicieux d'élaborer un modèle d'ascenseur simple dans le but d'avoir une première vue du projet. Ce modèle traite les requêtes selon leur ordre d'arrivée. D'autre part, l'ascenseur ne peut traiter qu'une requête à la fois.

1.1) Stockage et création des données.

Il nous est apparu judicieux de stocker les informations dans un type cellule d'une structure file.

```
typedef struct cellule // type cellule qui contient les informations
{
    long depart; // etage de depart de la requete
    long arrive; // etage d'arrive de la requete
    long tps;    // temps attendu avant le traitement
    long pris; // =0 si en attente , =1 si pris par l'ascenseur
    struct cellule* suivant; // adresse de la cellule suivante
} cellule;
typedef cellule * ptr_cellule;
```

```
typedef struct file // type file
{
    cellule* tete; // pointe sur la premiere requete
    cellule* queue; // pointe sur la derniere requete de la file
    long nb_elt; // nombre de requete dans la file
} file;
```

Ces données représentent le cœur du programme car c'est à partir de cette file et de ces cellules que le traitement s'effectue.

1.2) Fonction relative à la file

La première fonction relative à la file est l'initialisation avec la fonction *init*. Elle reçoit en entrée l'adresse de la file afin de mettre à jour tous les champs de la file.

```
void init(file *);
```

Par la suite, nous avons conçu une fonction qui représente la création d'un appel (une nouvelle requête). Cette fonction a l'entête suivant :

```
cellule* creer_requete(entier nb_etage)
```

Cette fonction reçoit en paramètre le nombre d'étage pour que l'on puisse faire une saisie sécurisée des informations de la nouvelle requête. En sortie nous retournons l'adresse mémoire où la requête est stockée.

```
cellule* creer_requete(entier nb_etage)
{
    cellule *re;
    entier reponse;
    FAIRE
    {
        afficher("entrer l'etage de depart ");
        saisir(reponse);

        } TANT QUE((reponse>nb_etage-1) ou (reponse<0)); //reponse doit être positif et inférieur
au nombre d'étage dans l'immeuble
        re->depart=reponse; // on stocke

        reponse=-3; //cas impossible

        FAIRE
        {
            afficher("entrer l'etage d'arrive ");
            saisir(reponse)

            } TANT QUE ((reponse>nb_etage-1) OU (reponse<0) OU (reponse==re->depart));
// securisation depart =arrive, pas négatif et pas supérieur à l'immeuble

        re->arrive=reponse;
        re->tps=0;
        re->suivant=NULL;
        re->pris=0; //pas encore raiter par l'ascenseur
        return re; // on retourne l'adresse
    }
```

Toutes les requêtes sont misent en attente dans la file. Nous avons donc utiliser les algorithmes d'enfiler et défiler avec de légère modification dans défiler. En effet, celle-ci retourne en sortie un entier, qui représente le temps écoulé entre l'entrée de la requête et sa sortie de la file.

La fonction ***enfiler*** reçoit en donnée modifiée la file dans laquelle les requêtes sont stockées, ainsi que l'adresse de la nouvelle requête.

```
void enfiler( file *, cellule *);
```

La fonction défiler reçoit en donnée modifiée la file car on lui retire la requête traitée. La sortie est de type entier car nous retournons le temps d'attente de la requête avant d'être traitée pour la création des statistiques.

```
long defiler(file *);
```

1.3) Fonctions de traitement

1.3.1) Fonction déplacement

La première fonction relative au traitement des requêtes est **déplacement**, cette fonction effectue le déplacement physique de l'ascenseur en fonction de la requête qu'elle doit traiter.

Déplacement reçoit en donnée la file contenant les requêtes, ainsi que la position actuelle de l'ascenseur (**num_etage**), et le nombre d'étage dans l'immeuble (**nb_etage**) car cette variable va être utilisée par des fonctions graphiques appelées dans **déplacement**.

<code>long déplacement(file *f,long num_etage,long nb_etage).</code>
--

Cette fonction teste si l'ascenseur traite une requête, dans ce cas le champ **pris** de la requête est à 1. Dans le cas inverse on teste la position de l'ascenseur par rapport à la requête (par rapport au champ **départ** de la requête), ainsi on détermine si l'ascenseur doit monter ou descendre (**num_etage++** ou **num_etage--**). D'autre part, nous testons également si l'ascenseur est à l'étage de départ de la requête, dans ce cas le champ pris sera initialisé à 1 (= pris 'vrai').

Si l'ascenseur traite une requête, le déplacement sera semblable au cas précédent. Cependant, le déplacement sera déterminé grâce au champ arrivé de la requête.

A la fin de la fonction nous retournons la position de l'ascenseur actualisée.

1.3.2) Fonction mode_pas_a_pas

Cette fonction n'est pas uniquement le mode pas à pas. En effet, c'est à l'intérieur de celle-ci que l'utilisateur choisit l'opération qu'il veut effectuer. De ce fait, **mode_pas_a_pas** reçoit toutes les données relatives au bon fonctionnement du programme.

<code>void mode_pas_a_pas(file *f,cellule *r,long num_etage,long nb_etage,file_stat *fstat,stat *rs)</code>

Elle a donc en donnée modifiée l'adresse de la file, un pointeur de cellule qui sert à la création de chaque nouvelle requête. La position de l'ascenseur stocké dans **num_etage**, mais également les données relatives aux statistiques. C'est-à-dire la file fstat, ainsi qu'un pointeur pour la création de nouvelle cellule pour les statistiques.

Puis en donnée nous avons le nombre d'étage de l'immeuble pour les fonctions graphiques essentiellement.

A l'intérieur de la fonction on crée des variables de type entier pour servir de compteur, ou encore une variable time qui sert à incrémenter le temps dans chaque requête.

Dans cette fonction on rentre dans une boucle *faire tant que* l'utilisateur saisit l'opération à effectuer, ensuite on rentre dans **switch case** avec toutes les opérations possibles. Ce protocole s'effectue tant que l'utilisateur n'entre pas 'q' (= quitter).

A l'intérieur du **switch case** nous avons 7 cas possibles :

- 'a' : affiche les requêtes contenu dans la file, lance la fonction **aff_file**.
- 'r' : saisie d'une nouvelle requête, lance la fonction **creer_requete**.
- 'h' : affichage de l'aide, lance la fonction **help**.
- 's' : saut à t+n, fonctionne traitement avec une boucle qui compte le nombre de fois qu'il lance traitement pour effectuer le saut.
- 'c' : mode continu, traite toutes les requêtes, lance la fonction **continu**.
- 't' : traitement pas à pas (t+1).
- 'q' : quitter le programme, **break**.

Traitement : cas 't'

Etudions à présent le cas 't' (traitement), qui constitue la base pour la réalisation du mode saut et continu.

Nous n'avons pas créer de fonction pas à pas, toutes les instructions pour accomplir cette instruction sont dans le cas 't'. Cependant, pour une meilleure présentation et organisation du code cette fonction pourrait être découpé en une fonction à part entière, ce qui est très facilement réalisable. Cela n'a pas était fait car par la suite nous nous sommes concentré sur le modèle compliqué.

Tout d'abord nous incrémentons la variable temps , puis dans chaque requête de la file nous incrémentons le champ tps (temps d'attente de la requête). Puis nous lançons la fonction **deplacement**. Ensuite nous testons si la première requête de la file (f->tete) est arrivée à destination. Si nous nous trouvons dans ce cas alors la cellule est défilée de la file puis nous actualisons les bases de données des statistiques.

Saut à t+n : cas 's'

Pour ce qui est du mode saut, celle-ci reprend les instructions de traitement. Cependant, l'utilisateur saisit le nombre de saut qu'il veut effectuer. Cette saisie est sécurisée pour qu'il ne puisse entrer un temps négatif (retour dans le passé impossible !!!). En revanche, s'il entre un temps supérieur à la réalisation de toutes les requêtes le temps sera incrémenté jusqu'à 't+saut'.

```
Pour (i=0 à i=saut, i++)  
{  
    Traitement (à t+1)
```

```
}
```

1.3.3) Fonction continu

Cette fonction reprend une nouvelle fois le fonctionnement de traitement, sauf que celle-ci est incluse dans une boucle qui lance le traitement tant que la file n'est pas vide ($f->nb_elt \neq 0$).

Cette fonction reçoit en donnée modifiée la file, pointeur de cellule à supprimer, la file contenant les statistiques ainsi que le pointeur qui permet d'enfiler la nouvelle statistique, la position de l'ascenseur et le temps. Nous avons également en donnée le nombre d'étage. En sortie nous renvoyons le temps actualisé.

```
long continu(file *f, cellule *r, long num_etage, long nb_etage, file_stat *fstat, stat *rs, long temps)
```

On trouve également dans le code d'autres fonction qui ne sont plus utilisées et qui pourrait donc être supprimées, tels que *tps_requete* qui permettait de calculer le temps nécessaire afin de traiter une requête, en soit un mode qui traite requête par requête et non pas à pas.

1.4) Fonction statistique

Partie dans l'engouement du stockage de données dans les files, nous avons stockés tous les temps de chaque requête traité dans une file. Cependant, nous aurions pu également utilisés de simple variable de type entier, comme nous l'avons fait dans le modèle complexe.

```
typedef struct stat // type cellule qui contient les informations
{
    long tps; // temps attendu avant le traitement
    struct stat* suivant; // adresse de la cellule suivante
} stat;
typedef stat * ptr_stat;
```

```
typedef struct file_stat // type file
{
    stat* tete; // pointe sur la premiere requete
    stat* queue; // pointe sur la dernire requete de la file
    long nb_elt; // nombre de requete dans la file
} file_stat;
```

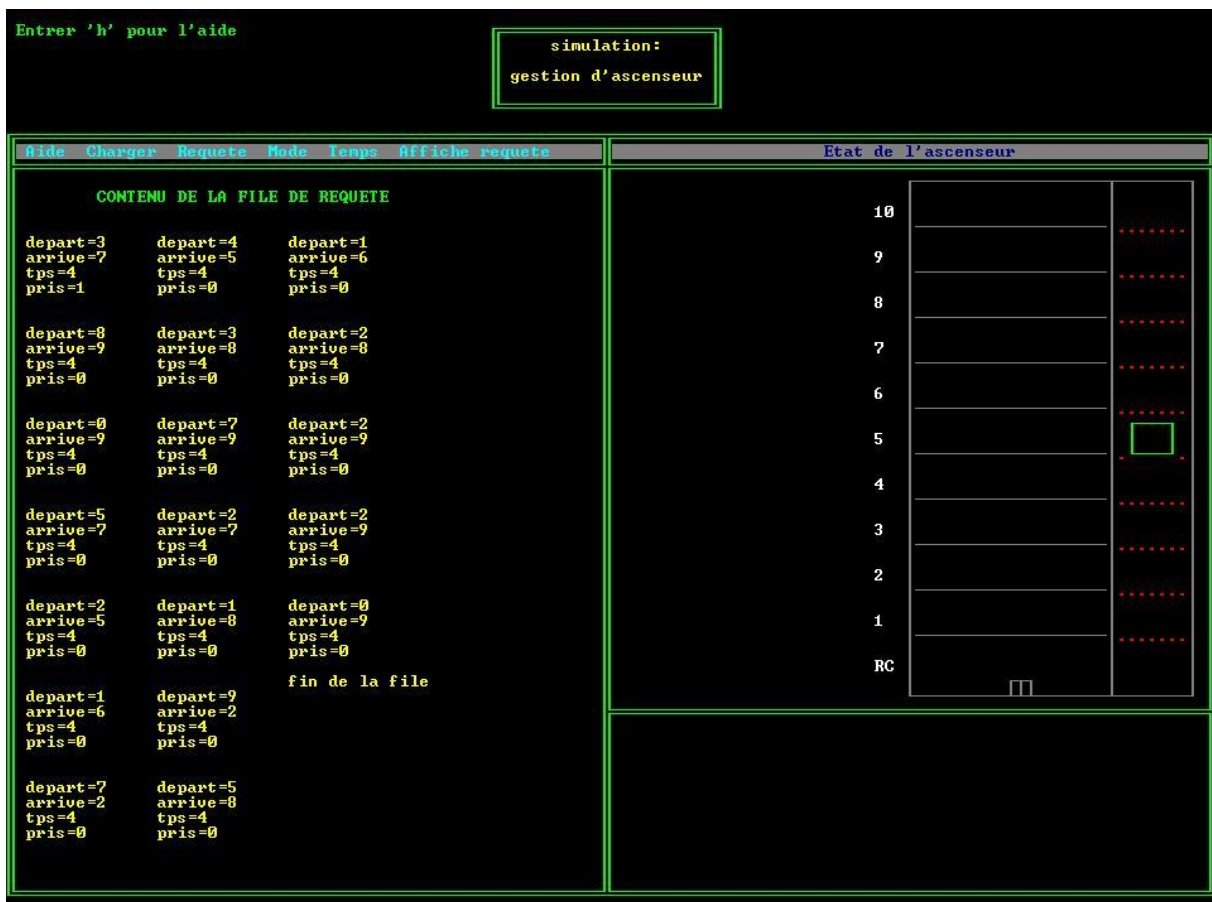
Pour calculer les temps moyen de traitement, nous avons conçu la fonction *calcul_stat*. Celle-ci reçoit en entrée la file statistique. Ensuite dans une variable temporaire nous stockons la somme des temps de chaque cellule. Puis nous divisons cette somme par le nombre d'élément dans la file, se résultat sera par la suite affiché dans le cadre prévu à cette effet.

1.5) Interface homme/ machine

Il nous est apparu évident qu'une interface graphique était indispensable au contrôle du programme. L'interface se divise en trois fenêtres. Dans la première l'utilisateur entre ses opérations, nous effectuons aussi l'affichage de toutes les données que l'utilisateur peut avoir besoin, toutefois ces données ne sont pas affichées en permanence, elles s'affichent à la demande de l'utilisateur. Dans la seconde fenêtre, nous affichons l'immeuble, ainsi que l'ascenseur. La dernière fenêtre est utilisée pour l'affichage des statistiques. Cette interface graphique a été réalisée pour le modèle simple, mais a été pensée pour être facilement intégré au modèle complexe.

Nous avons tout d'abord une fonction qui affiche le cadre général, et les sous fenêtres. Une seconde fonction qui dessine l'immeuble, dans celle-ci nous pouvons choisir le nombre d'étage. Puis une dernière fonction qui dessine l'ascenseur selon sa position dans l'immeuble, grâce à cette fonction nous pouvons modéliser visuellement le déplacement de l'ascenseur à chaque temps.

Ensuite, nous avons également réalisé des fonctions qui effacent le contenu de chaque fenêtre afin d'éviter les bugs graphiques. Pour la réalisation de cet affichage nous avons utilisé la librairie *conio.h*, et tout particulièrement les fonctions *gotoxy*, *textcolor* et *textbackground*.



2) Architecture du projet : Modèle complexe

Dans ce modèle, nous avons réfléchi à une façon d'optimiser le programme pour minimiser le nombre d'opérations qu'il effectue à chaque traitement, donc de réduire son temps d'exécution. Le stockage des requêtes est fait selon une architecture plus complexe que pour le 1^{er} modèle mais qui facilite la recherche et l'exécution des fonctions de gestion.

Nous fonctionnons avec un « immeuble », sorte de banque qui stocke toute les requêtes en attente et un ascenseur, sorte de banque qui stocke les requêtes en cours de traitement. Ces deux banques sont de type « file » que nous définirons plus tard. Pour charger la banque « immeuble » nous avons la possibilité de rentrer des requêtes manuellement ou de les charger à partir d'un fichier texte. Pour ce qui est de la « banque » ascenseur, elle peut contenir plusieurs requêtes car l'ascenseur peut traiter plusieurs requêtes en même temps. C'est une amélioration par rapport au modèle simple qui traite juste requête par requête. L'utilisateur peut ensuite choisir son mode de traitement, pas à pas (cycle par cycle) ou continue (n cycle jusqu'à ce que toutes les requêtes soient traitées). Un cycle de traitement s'effectue en plusieurs phases :

1&2-Recherche dans l'immeuble de requêtes qui satisfont les critères de validité

3-Chainage des requêtes valides au module « ascenseur »

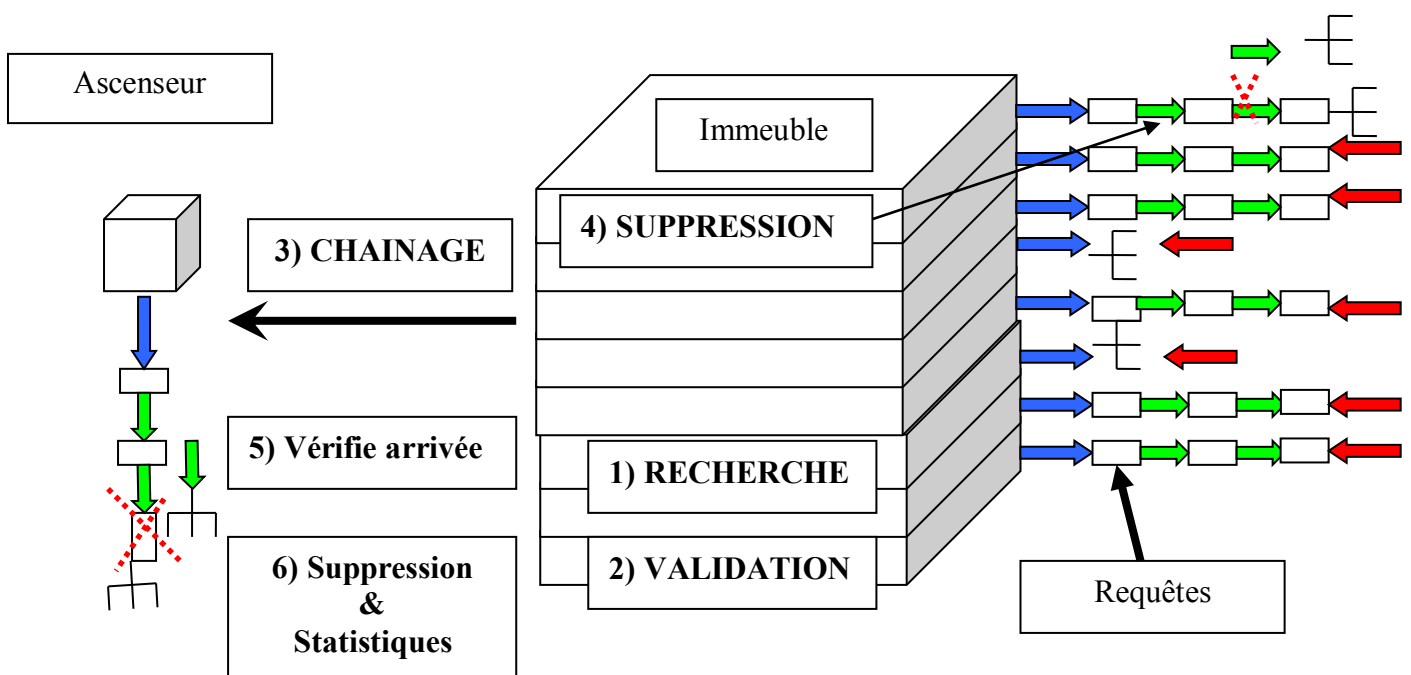
4-Suppression de ces requêtes de la banque « immeuble »

5-Recherche dans l'ascenseur des requêtes finies de traitées

6&7-Statistiques sur le temps de traitement de ces requêtes et suppression des requêtes ainsi que de leur zone mémoire

8-Incrementation de la position de l'ascenseur et du temps (déplacement)

Fonctionnement d'un cycle de traitement : schéma explicatif



Légende :

-  Pointeur suivant
-  Pointeur fin de file
-  Pointeur début file
-  Requête (élément de base des files)
-  NULL

2.1) Types de fichiers utilisés

Nous avons décidé pour ce projet de séparer les fonctions de traitement et l’affichage graphique. Nous avons donc mis nos fonctions de traitement dans un fichier nommé fonctions.c et nos fonctions graphiques dans le fichier nommé affichage.c. La création d’un fichier fonctions.h avec les entête de toutes les fonctions a été nécessaire et est intégrée dans notre fichier main.c qui contient la structure principale du programme.

2.2) Définition des types utilisés

Pour ce modèle nous avons défini deux types, qui nous permettent de traiter plus facilement les données.

Le type requette

Ce type permet de stocker les données d’une requête et est défini comme ceci :

```
typedef struct requette    //structure requette
{
    long etage_init;        //étage ou la requette est formulée
    long etage_voulu;       // étage de destination de la requette
    long time;              //temps auquel est formulée la requete
    struct requette* suivant; //pointeur vers un type requette (pour le stockage)
}requette;
```

C’est l’unité de stockage la plus simple, équivalente à un maillon, on l’utilise ensuite dans des structures plus complexes.

Le type file

Ce type permet le stockage des requêtes (voir ci-dessus), cette structure est une chaîne d’élément de type « requette » et comporte un pointeur vers son 1^{er} élément et un vers son dernier. Elle stocke aussi son nombre d’éléments constituant la chaîne.

2.3) Architecture détaillée

Nous décomposerons l'architecture en plusieurs parties. Tout d'abord nous allons définir les variables déclarées dans le « main » ainsi que leur fonctions. Ensuite nous expliquerons comment sont créés les données utilisateur pour le fonctionnement de l'ascenseur. Nous parlerons ensuite du stockage de celle-ci en justifiant nos structures et en développant les fonctions qui les mettent en jeu. Enfin nous terminerons avec les fonctions de traitement des données et faisant fonctionner le programme en lui-même.

2.3.1) Variables utiles du main

Entier « Nb_etages » : Cette variable stocke le nombre d'étages de l'immeuble. Nous avons choisi de stocker ce paramètre dans une variable pour que notre modèle puisse s'adapter à différents immeubles. Nous l'avons initialisé à 10 étages pour des raisons d'affichage, mais nous considérons qu'il peut être initialisé jusqu'à 15 étages. Au delà notre modèle ne sera plus optimisé et le temps de traitement sera trop long. De plus dans la réalité les immeuble d'une taille supérieur à 15 étages ont généralement plusieurs ascenseurs.

Entier « Time » : Cette variable stocke le temps « absolu » c'est-à-dire le temps du programme, elle est initialisé à 0 et incrémenté après chaque cycle de traitement.

Entier « Pos ascenseur » : Elle donne la position de l'ascenseur, initialisée a 0 au début du programme, elle est modifié à chaque déplacement.

Entier « Temps_total » : Stocke le temps total de traitement des requêtes, initialisé à 0 en début de programme, on y ajoute le temps de traitement d'une requête à chaque fois qu'une requête est finie de traitée.

Entier « Nb_total » : Stocke le nombre de requêtes traitées par le programme, initialisé à 0 en début de programme, on y ajoute 1 à chaque fin de traitement d'une requête. Avec la variable « temps_total » elle permet de calculer le temps moyen de traitement d'une requête.

File « ascenseur » et file « immeuble[] » sont des structures de stockage des données qui seront détaillées dans le suite de ce rapport.

2.3.2) Création des données

Les seules données créées par l'utilisateur sont les requêtes, elles peuvent être créées une par une ou importées à partir d'un fichier texte.

2.3.2.1) Parseur

La fonction `parseur(char *nom, file immeuble[])` permet d'importer des données à partir d'un fichier texte. Elle prend en paramètre le nom du fichier texte ainsi que la « banque » immeuble pour y ajouter les nouvelles requêtes.

Le fichier texte doit contenir des informations sous la forme :

Temps de la requête étage de départ-etage de destination

La fonction va repérer le temps grâce au séparateur « », ensuite l'étage de départ grâce au séparateur «-» ainsi que l'étage d'arrivée.

Elle initialisera une requête avec ces informations, puis la chaînera à l'immeuble grâce à la fonction «`chainage_immeuble`» (fonction définie plus tard). Le parseur utilise tout le temps le même espace mémoire pour stocker les informations des requêtes qu'il importe du fichier texte. Ainsi lors de la création d'une deuxième requête les informations de la 1^{ère} seront écrasées dans la variable de type « requête » du parseur. Ceci ne pose aucun problème car la fonction «`chainage_immeuble`» alloue un nouvel espace mémoire et y recopie la requête qu'elle prend en entrée avant de la stocker dans l'immeuble.

Enfin le parseur recommencera ces opérations tant que le fichier texte n'aura pas été intégralement parcouru, chaînant ainsi toutes les requêtes à l'immeuble.

2.3.2.2) `creation_requette_manuel`

La fonction `creation_requette_manuel(file immeuble[], long nb_etages, long time)` permet à l'utilisateur de créer une nouvelle requête au cours de l'exécution du programme. Celui-ci saisit l'étage où il veut aller et l'étage où il se trouve. Nous avons sécurisé ces saisies pour que les étages correspondent à des étages existants (c'est pour cela qu'on entre le nombre d'étages de l'immeuble en paramètre). Une sécurisation a aussi été faite pour que l'utilisateur n'entre pas une requête qui a pour destination l'étage où elle est formulée.

Le temps pris en compte est celui auquel est formulé la requête (paramètre «`time`» de la fonction), nous avons créé un modèle où l'utilisateur pouvait rentrer le temps où est créée la requête mais suite à de nombreux conflits nous avons préféré ne pas le faire saisir par l'utilisateur.

Nous initialisons ensuite les champs d'une requête (de type « requête ») avec ces informations. Nous effectuons ensuite un appel à la fonction «`chainage_immeuble`» qui alloue la mémoire nécessaire et chaîne la requête à la banque de requête en attente (« immeuble »).

2.3.3) Structures et fonctions de stockage des données

Immeuble

C'est ici que nous stockons les requêtes en attente. Nous utilisons une file de requête par étage, regroupées dans un tableau que nous déclarons dans le programme principal. Nous avons donc un tableau de files, avec autant de files que d'étage dans notre immeuble, appelé `immeuble[]`. Lors de l'entrée d'une requête par l'utilisateur ou directement importée depuis un fichier texte, la file du tableau `immeuble` correspondant à l'étage où la requête est formulée sera modifiée. Cette architecture nous permet donc de savoir rapidement s'il y a des requêtes en attente et ce pour chaque étage, avec le champ `"nb_elt"` de chaque files.

Fonctions relatives a l'immeuble

2.3.3.1) `init immeuble`

La fonction `init_immeuble(ptr_file immeuble[])` nous permet d'initialiser l'immeuble en position vide, tout les champs « `nb_elt` » des files sont initialisés à 0 et tout les pointeurs « `debut` » et « `fin` » des files sont initialisés à `NULL`. Cette fonction est appelée au début du programme.

2.3.3.2) `chainage_immeuble`

La fonction `chainage_immeuble(file immeuble[] , requette nouvelle_requette)` nous permet de stocker une requête qui viens d'être formulée dans l'immeuble. Elle reçoit en entrée l'immeuble (tableau de file) et une requête (de type « `requette` »).

La fonction alloue un espace mémoire d'une requête et initialise ses champs avec les informations contenues dans la requête que la fonction reçoit en entrée. Nous avons préféré réserver l'espace mémoire nécessaire directement dans la fonction plutôt que de lui donner en entrée un pointeur vers une requête pour éviter les conflits avec la fonction « `parseur` ». Ensuite on repère à quel étage est formulée la requête (champ « `etage_init` ») et on l'ajoute à la file de l'immeuble correspondante, cet ajout est en réalité un enfilage. On incrémente aussi le champ « `nb_elt` » de la file où a été chaînée la requête.

Si une requête est formulée à l'étage 4 par exemple, on l'enfilera dans la file à l'indice 4 du tableau `immeuble` et ajoutera +1 à « `nb_elt` » de la file pour indiquer qu'elle contient une requête de plus.

2.3.3.3) `suppression_immeuble`

La fonction `suppression_immeuble(file immeuble[], ptr_requette re )` permet de « `supprimer` » une requête (paramètre « `re` ») de l'immeuble. On notera que la zone mémoire de la requête n'est pas libérée, car une fois enlevée de l'immeuble la requête doit être traitée et donc ses informations sont utilisées par d'autres fonctions. En effet les requêtes situées dans l'immeuble sont celle en attente de traitement, la suppression de l'immeuble indique qu'elle sera en cours de traitement.

Pour qu'elle n'apparaisse plus dans l'immeuble on change juste les pointeurs de la file étage ou elle se trouvait et on enlève 1 au champ « nb_elt » de la file pour indiquer qu'il y a un élément en moins dans cette file.

2.3.3.4) elt_immeuble

La fonction elt_immeuble(file immeuble[], long nb_etages) est utilisée pour le mode continu, elle effectue un parcours de l'immeuble et vérifie les champs « nb_elt » de chaque files d'étages. Elle s'arrête une fois qu'elle a finit de parcourir tout l'immeuble (c'est pour cela qu'on entre en paramètre « nb_etages »), ou quand elle trouve un élément. Elle retourne un booléen qui permet de savoir si il reste des requêtes en attente.

Ascenseur

L'ascenseur est une file de requêtes, elle stocke les requêtes en cours de traitement. On y enfile toutes les requêtes valides. On contrôle donc dans la banque « immeuble » les requêtes valides à chaque temps et on les chaîne à l'ascenseur.

L'ascenseur traite la 1^{ère} requête de sa file, toutefois une vérification à chaque étage de la totalité des requêtes situées dans la file est effectuée, au cas ou une requête chaînée après la 1^{ère} soit arrivée à destination, auquel cas on la supprimera de l'ascenseur.

Fonctions relatives à l'ascenseur

2.3.3.5) init_ascenseur

La fonction init_ascenseur(ptr_file ascenseur) initialise l'ascenseur en position de départ, c'est-à-dire que son nombre d'élément est à 0 et ses pointeurs début et fin sont à NULL. On appelle cette fonction en début de programme.

2.3.3.6) chainage_ascenseur

La fonction chainage_ascenseur(ptr_file ascenseur, ptr_requette nouvelle_requette) enfile une requête à l'ascenseur. Elle est différente de la fonction « chainage_immeuble » car ici on entre en paramètre un pointeur vers une requête (« nouvelle_requette »). Effectivement la requête est déjà créée et stockée en mémoire, on l'enfile donc directement à l'ascenseur. Lors de l'exécution du programme, cette fonction chaîne une requête qui était en attente dans la banque « immeuble » et qui a été défini comme valide. Ensuite on appelle la fonction « suppression_immeuble » qui va changer les pointeurs pour faire disparaître cette requête de l'immeuble.

Contrairement à l'immeuble, on ne dispose pas ici de fonction qui supprime un élément de l'ascenseur, cette suppression est effectuée dans une des fonctions de traitement.

2.3.4) Fonctions de traitement des données

2.3.4.1) Sélection et validation des requêtes en attente

La fonction `gestion_requette(file immeuble[], ptr_file ascenseur, entier nb_etages, entier pos_ascenseur, entier time)` permet de chercher les requêtes à chaîner à l'ascenseur selon des critères de validité variables. Ces critères diffèrent selon que l'ascenseur est vide ou plein, on a donc un test qui se fait et on appelle la fonction correspondante (« `gestion_vide` » ou « `gestion_plein` »).

Cas de l'ascenseur vide

Dans ce cas on appelle la fonction `gestion_vide(file immeuble[], ptr_file ascenseur, entier nb_etages, entier pos_ascenseur, entier time)`. Elle effectue une recherche dans le tableau `immeuble` pour trouver la requête la plus proche de la position de l'ascenseur, pour ce faire elle utilise la variable « `pos_ascenseur` » qu'elle reçoit en entrée.

On regarde tout d'abord si il y a une requête à l'étage courant de l'ascenseur, auquel cas on chaîne directement la 1ere requête de la file à l'ascenseur et on appelle la fonction « `gestion_plein` » chaîner toute les autres requêtes de la file étage qui ont le même sens que celle chaînée dans l'ascenseur.

Si il n'y a pas de requêtes à l'étage courant de l'ascenseur on fait une recherche au dessus et au dessous de l'ascenseur et on stocke dans des pointeurs les premières requêtes que l'on trouve. Ensuite on compare les deux pointeurs pour déterminer où se trouve la requête la plus proche (au dessus ou au dessous de l'ascenseur). Enfin on chaîne une requête "virtuelle" qui a pour étage de destination l'étage où a été formulée la requête en attente la plus proche. En effet pour que l'ascenseur puisse traiter une requête située au dessus ou en dessous de lui, il faut d'abord qu'il se rende à l'étage où elle a été formulée, c'est pourquoi nous utilisons cette requête « virtuelle ».

Cas de l'ascenseur plein

On repère d'abord le sens de la requête que l'ascenseur est en train de traiter (la 1ere de la file ascenseur) pour savoir si celui ci monte ou descend. Ensuite on parcourt la file étage correspondant à la position courante de l'ascenseur et on chaîne à l'ascenseur toutes les requêtes de la file étage qui ont le même sens que la requête en cours de traitement.

Lors de tout ces traitements on prend en compte le temps général (qui est stocké dans la variable « `time` » déclaré dans le main) et on ignore toute les requêtes en attente dans l'immeuble qui ont un temps supérieur à « `time` ». En effet ces requêtes sont physiquement stockées en attente mais n'existent pas encore, elles seront prises en compte quand leur temps sera inférieur ou égal au temps du main.

2.3.4.2) requette_fini

La fonction `requette_fini(ptr_file ascenseur, entier pos_ascenseur, entier time, entier* temps_total, entier* nb_total)` permet de vérifier dans la file ascenseur si des requêtes sont finies de traitées. On compare donc l'étage courant de l'ascenseur (« `pos_ascenseur` » passé en paramètre) à l'étage de destination de chaque requête (champ « `etage_voulu` » des requêtes). Si une des requêtes est arrivée à destination on la supprime de la file en changeant les pointeurs de celle-ci. On enlève ensuite 1 au champ « `nb_elt` » qui stocke le nombre d'éléments de la file, et on libère l'espace mémoire que la requête occupait. On calcul son temps de traitement : temps actuel (« `time` » du main passé en paramètre à la fonction) moins le temps où a été formulé la requête (stockée dans le champs « `time` » de la requête). On stocke ce temps dans une variable que l'on envoie en paramètre à la fonction de statistiques. Les variables « `temps_total` » et « `nb_total` » ne sont pas utilisées par la fonction « `requette_fini` » en elle-même mais dans une fonction que celle-ci appelle, c'est pourquoi nous les lui passons en paramètre.

2.3.4.3) Escalier

La fonction `escalier(file immeuble[], entier time, entier nb_etages)` nous permet de supprimer les requêtes trop vieilles de la banque des requêtes en attente (stockée dans le module « `immeuble` » passé en paramètre). Notre modèle ne gérant pas les priorités temporelles, nous avons décidé de supprimer les requêtes trop vieilles. Sont considéré comme telles les requêtes en attente depuis plus de 48 temps. Nous avons considéré 1 temps = 1 déplacement de l'ascenseur, moyennant 10 secondes environs par déplacement, 48 temps correspond à 4 min et personne n'attend un ascenseur plus de 4 min.

Cette fonction parcourt donc l'immeuble et compare tout les temps où ont été formulé les requêtes en attente (stockées dans le champ « `time` » des requêtes) au temps global (« `time` » du main, passé en paramètre dans la fonction).

Si une requête est détectée comme trop vieille on appelle la fonction « `suppression_immeuble` » détaillée ci-dessus, puis on libère l'espace mémoire où était stockée la requête.

2.3.4.4) Deplacement

La fonction `deplacement(ptr_file ascenseur, entier* pos_ascenseur, entier* time, entier nb_etages)` incrémente le temps et la position de l'ascenseur (passés en paramètre) en fonction de la requête qu'il traite. Si l'ascenseur monte, il va augmenter la position de l'ascenseur, si il descend il va diminuer sa position de 1, enfin si il est vide il incrémentera juste le temps (passé en paramètre). Nous passons les adresses des variables « `pos_ascenseur` » et « `time` » car on le peut les retourner toute les deux en fin de fonction. On modifie donc directement l'information à leur adresse pour ne pas avoir à les retourner en fin de fonction.

2.3.4.5) Stats

La fonction stats(entier nouveau_temps, entier* temps_total, entier* nb_total) rajoute le temps de traitement d'une requête qui viens d'être finie à la variable « temps_total » (déclarée dans le main) qui stocke le temps de traitement de toute les requêtes et incrémente la variable « nb_total » qui stocke le nombre de requêtes traitées. Le calcul du temps moyen de traitement n'est pas effectué dans cette fonction, on actualise seulement les éléments qui permettent de le calculer.

2.3.5) Affichage

Pour l'affichage général (cadre et ascenseur) nous utilisons les mêmes fonctions que pour le modèle simple car nous avons réfléchis celle-ci pour qu'elles s'adaptent au modèle complexe. Nous avons toutefois codé d'autres fonctions qui affichent des éléments utiles à l'utilisateur.

2.3.5.1) affichage_requette_attente

La fonction affichage_requette_attente(file immeuble[], entier nb_etages) parcourt tout l'immeuble et affiche le nombre de requêtes en attente à chaque étage (en utilisant l'information « nb_elt » de chaque file étages). Si il y a des requêtes en attente on affiche leur étage de destination (champs « etage_voulu » des requêtes).

2.3.5.2) affichage_ascenseur

La fonction affichage_ascenseur(ptr_file ascenseur) affiche tout les requêtes en cours de traitement dans l'ascenseur avec leur étage de départ et leur étage de destination (stockées dans les champs de la requête).

2.3.5.3) petit_cadre

La fonction petit_cadre(entier temps_total, entier nb_total, entier time, ptr_file ascenseur, entier pos_ascenseur) affiche dans le petit cadre des informations de contrôle que l'utilisateur peut voir en permanence. On y affiche le temps, la position de l'ascenseur la requête en cours de traitement et le temps moyen de traitement d'une requête. Toutes ces informations sont passées en paramètre de la fonction, seul le temps moyen de traitement est calculé au cours de l'exécution de la fonction.

3) Problèmes rencontrés // Méthode

Nous travaillons toujours ensemble chez l'un ou l'autre, ainsi nous pouvons résoudre nos problèmes ensemble. Ce système est pratique car nous pouvons discuter de notre vision du projet et déterminer ensemble le fonctionnement de celui-ci et donc partir sur la même base avec la même perspective. Pour ce qui est de la répartition nous avons déterminé les fonctions nécessaires au projet et nous nous les sommes partagées. Comme nous avons travaillé ensemble il était facile de faire des fonctions qui s'intègrent à l'ensemble du projet. Ensuite nous avons procédé en testant chaque fonction individuellement avec toutes les combinaisons possibles pour la valider. Ainsi nous pouvions être sur que nos fonctions marchaient correctement et ce pour toutes les combinaisons d'entrées possibles, ce qui permettait de l'insérer sans problèmes dans le programme.

Les premiers problèmes que nous avons rencontrés concernent la détermination de tous les cas possibles pour l'exécution de chaque fonction, ainsi que le fait que certaines requêtes n'existent pas encore (car formulées à un temps postérieur à celui du programme). Nous avons donc imaginé des modes de traitement qui prennent en compte le temps où est formulée la requête.

Nous avons aussi passé beaucoup de temps à essayer de résoudre des problèmes de mémoire dus aux pointeurs. Le problème principal étant une modification automatique des pointeurs suivant des éléments de l'immeuble lors d'allocations mémoire. L'allocation n'a aucune relation avec l'immeuble, mais la modification s'opère parfois et engendre un plantage dans la suite du programme. La difficulté pour résoudre ce bug réside dans le fait qu'il n'apparaît pas tout le temps et ce pour les mêmes combinaisons de requêtes. Nous avons finalement décidé de le laisser en le signalant dans ce rapport, mais nous ne comprenons toujours pas pourquoi cette modification a lieu.

Pour essayer de résoudre nos bugs nous avons utilisé une fonction « log » qui crée un fichier texte dans lequel elle met les affichages de contrôle de l'exécution du programme. Ainsi nous pouvons voir exactement où le programme a planté même si la fenêtre où il s'exécute se ferme toute seule. Nous avons laissé les affichages de contrôle dans le programme final pour que le correcteur puisse visualiser où se trouvent les bugs (dus aux pointeurs voir ci-dessus).

Conclusion

En conclusion on peut dire que ce projet nous a beaucoup apporté sur le plan de la réflexion. Il nous a permis de voir comment envisager différents systèmes pour le même but, avec un souci d'optimisation. Ainsi nous avons réalisé deux modèles qui gèrent les données différemment, le deuxième modèle étant plus optimal dans le traitement des requêtes mais aussi dans le nombre d'opération qu'il fait pour le traitement (avec l'architecture de stockage que nous avons utilisé ce nombre est minimal).

Nous avons aussi rencontré de nombreux problèmes avec les pointeurs et la mémoire, problèmes obscurs que nous n'avons pu résoudre. Nous avons pu prendre conscience que certains problèmes sont inexpliqués, car résultant de problèmes en mémoire, et que nous ne pouvons les résoudre à notre niveau.