Projet d'algorithmique

Cours d'Informatique de Deuxième Année

—Deug Sciences M3-MI—

Gestion d'un ascenseur

- ce travail doit être rendu au plus tard le lundi 5 janvier au secrétariat des DEUG Sciences, la soutenance aura lieu lors de la dernière séance de Travaux Pratiques (le jeudi 8 janvier) ;
- vous devrez vous inscrire sur la liste de soutenance disponible au secrétariat en même temps que vous rendrez le projet ;
- la clarté et le soin apportés à la rédaction de votre travail, ainsi que la lisibilité de votre *listing* seront pris en compte ;
- ce travail peut être réalisé en binôme, chaque personne devant être présente à la soutenance et pouvant être interrogée individuellement ;
- la partie programmation devra être réalisée en *C ANSI*, compilable directement sous *HP–UX* (l'*UNIX* de *HP*).

---

1 Présentation du sujet

Le projet de ce semestre consistera en la réalisation d'un logiciel de simulation du fonctionnement d'un ascenseur (voire de plusieurs ascenseurs). Tout le monde a, au moins une fois dans sa vie, utilisé cet outil dont l'invention à la fin du siècle dernier, a considérablement modifié l'architecture de nos cités (pourquoi ?). Les personnes souhaitant obtenir plus d'informations à ce sujet sont invités à consulter le serveur **Web** de la société **OTIS** (1).

La grosse difficulté de ce projet consiste à donner un *modèle* pour simuler le fonctionnement d'un ascenseur. Nous allons considérer notre ascenseur (ou notre ensemble d'ascenseurs) comme un système évoluant au cours du temps. Se donner un modèle, c'est se donner un ensemble de paramètres permettant de décrire l'évolution de notre système suivant les contraintes que nous allons lui imposer.

Afin de simplifier le problème, nous proposons de définir un modèle *discret* de simulation. Cela signifie que nous supposons que tous les changements dans l'état du système (c-à-d. les variables de notre système) se passent à un instant discret du temps. On observe l'évolution du système à intervalle de temps régulier. Cet intervalle correspond au temps du déplacement de l'ascenseur entre 2 étages. A chaque top d'horloge, nous autorisons les évènements suivants :

- on prend en compte les nouvelles *requêtes* (*e.g.* Ernest appuie sur le bouton) ;

(1) http://www.otis.com

- on traite la requête de priorité maximale (nous verrons comment caractériser cette requête).

Dans la suite, nous allons voir comment appliquer le modèle précédent pour simuler le fonctionnement d'un ascenseur.

2 Modélisation

2.1 Un modèle simple

Dans cette partie, nous introduisons un modèle simplifié. Par la suite, nous ferons des modifications sur les contraintes de ce modèle afin de rendre le modèle plus réaliste.

Nous considérons un immeuble ayant 3 étages (nous désignerons le rez-de-chaussé par 0). Cet immeuble est muni d'un seul ascenseur, chaque étage disposant d'un bouton d'appel pour l'ascenseur. On impose les contraintes de fonctionnement suivantes :

- on ne distingue pas les appels pour monter ou descendre ;
- l'ascenseur est soit en attente d'une requête (donc en position d'arrêt), soit en cours de traitement d'une requête (en montée ou en descente) ;
- on ne peut pas arrêter l'ascenseur en cours de trajet.

Nous illustrons le modèle précédent par l'exemple de la figure 1.

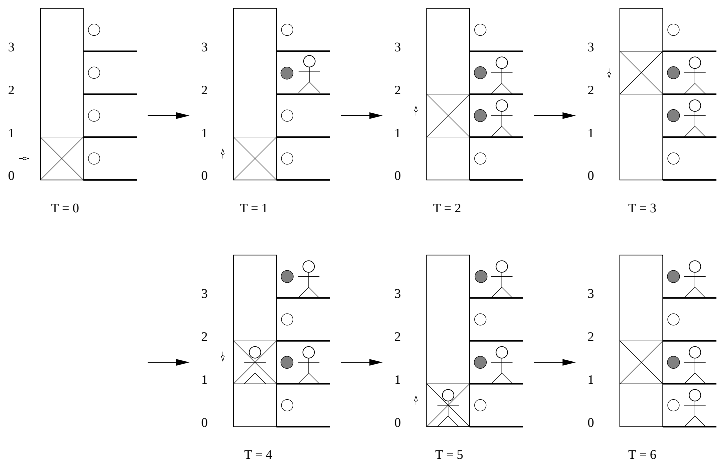

Figure 1: Un modèle

Dans ce modèle, les requêtes sont gérées suivant l'algorithme suivant :

- les requêtes sont classées suivant leur ordre d'arrivée ;
- une requête devient prioritaire si son étage correspond à l'étage courant de l'ascenseur (*i.e.* une requête à l'étage `i` est prise en compte si l'ascenseur est en arrêt à cet étage).

Ces deux contraintes imposent d'utiliser la structure de données suivante. Nous définissons une *file de priorité* par étage. Chaque file étant gérée suivant l'ordre d'arrivée des requêtes. La structure la plus adaptée pour implémenter une file de priorité est une liste chaînée. Chaque cellule contiendra les informations relatives à une requête (appel d'un individu à un étage donné). Le problème se ramène donc à insérer une nouvelle requête dans la file de priorité. Ainsi, a chaque instant, la requête de priorité la plus forte sera en tête de la file.

On pourra utiliser les types suivants :

```
struct cell { info mvt;              /* info est un type (choisi par vos */
                                     /* soins) qui permet de stocker les */
                                     /* informations relatives a une     */
                                     /* requete.                         */
              struct cell * suivant };

typedef struct cell Cellule;
typedef Cellule * Liste;             /* file pour un etage donne         */
Liste * Immeuble ;                   /* tableau de pointeurs             */
```

Hormis le fait de voir l'évolution des déplacements dans un immeuble, il peut être intéressant de vérifier l'efficacité de notre modèle. Ainsi, on peut ête amené à définir un autre modèle.

2.2 Autre modèle

Dans le modèle précédent, il était impossible de stopper l'ascenseur pour traiter deux requêtes simultanément. Par exemple, dans la session présentée en section 3.1, il aurait été préférable de s'arrêter à l'étage `3` pour gérer l'appel `3->7`.

Une amélioration consisterait donc à pouvoir gérer plusieurs requêtes simultanément. Pour cela, nous imposons de nouvelles contraintes :

- on distingue les appels montants et descendants ;
- on stoppe l'ascenseur (par exemple en cours de montée) si et seulement si une nouvelle requête compatible avec son mouvement apparaît.

Il semble évident que ce nouveau modèle soit plus efficace que le précédent. Ceci devra être vérifié par une série de tests, ou par des statistiques...

3 Description du travail à rendre

Le programme que vous devez faire doit être interactif, au sens où l'utilisateur doit pouvoir faire des requêtes et obtenir des réponses. On pourra donc prévoir un petit langage de commandes permettant à l'utilisateur de passer ses demandes au programme. Le langage utilisé devra être très simple et pourra s'inspirer de l'exemple suivant (minimal).

Au lancement de `a.out` (par exemple), le programme affiche un *prompt*—nous avons choisi `Lift>`—et se met en attente d'un ordre de la part de l'utilisateur. La première commande utile sera la commande `h` fournissant de l'aide sur les commandes disponibles.

```
$a.out
Lift> h
```

Commande	*Effet*
`h`	`Pour obtenir cette aide`
`o nom`	`Chargement du fichier contenant les requêtes`

`r <d> <a>`	`Ajout d'une requête :`
	`- d : étage de depart`
	`- a : étage d'arrivée`

`v`	`Visualisation des requêtes non traitées`
`V`	`Visualisation des requêtes en cours de traitement`

`m <n>`	`Selectionne un mode de fonctionnement`
	`- 0 : mode continu`
	`- 1 : mode pas a pas`

`t <n>`	`Saute de t à t+n. Si n n'est pas precisé` `le saut est de 1 si on est en mode pas a pas`
`S`	`Edition de statistiques`
`q`	`Quitter`

3.1 Exemple de session

```
$a.out
Lift:0>r 2 5
Lift:0>v
Ascenseur a l'etage 0
Appel etage 2 vers 5
Lift:0>V
Ascenseur etage 0 : aucune action en cours
Lift:0>t
Lift:1>V
Ascenseur etage 1 : (2->5)
Lift:1>v
Ascenseur a l'etage 1
Pas de requete
Lift:1>r 3 7
Lift:1>t 2
Lift:3>V
Ascenseur etage 3 : (2->5)
Lift:3>v
Ascenseur a l'etage 3
Appel etage 3 vers 7
Lift:3>t 2
t=5 : requete (2->5) terminee
Lift:5>V
Ascenseur etage 5 : aucune action en cours
```

```
Lift:5>v
Ascenseur a l'etage 5
Appel etage 3 vers 7
Lift:5>m 0
Lift:5>t
t=11 : requete (3->7) terminee
Lift:11>V
Ascenseur etage 7 : aucune action en cours
Lift:11>v
Ascenseur a l'etage 7
Pas de requete
Lift:11>S
2 requetes traitees
```

L'exemple de session précédente montre qu'il est fastidieux de tester le programme “en ligne”. Afin de rendre ces tests moins rébarbatifs, nous allons définir un “langage simple” pour exprimer les événements. Le programme que vous devez faire doit lire les requêtes dans un fichier écrit dans le format suivant :

- une requête sera codé par une chaine de 3 caractères ;
- une série de requêtes relatif à un instant `t` est représentée sur une même ligne.

Par exemple, les deux premières lignes du fichier suivant contiennent les requêtes traitées dans la session décrite en 3.1 :

```
00 2-5
01 3-7
13 3-5 4-1 7-0
```

La troisième ligne du fichier sera interprétée par :

- appel de l'étage `3` vers l'étage `5` au temps `t=13` ;
- appel de l'étage `4` vers l'étage `1` au temps `t=13` ;
- appel de l'étage `7` vers l'étage `0` au temps `t=13`.

4 Améliorations et options

Les améliorations et les options ne sont pas obligatoires, mais constituent un *plus* pour l'évaluation finale de votre travail. Vous êtes invités à faire part de vos propositions aux enseignants pour savoir dans quelle mesure elles sont réellement pertinentes.

On peut envisager les options suivantes :

- la réalisation d'un affichage plus convivial ;

- la limitation de la capacité de l'ascenseur ;
- l'hypothèse d'une seul ascenseur est très restrictive. On peut envisager l'emploi de plusieurs ascenseurs. Il est donc important d'expliquer vos choix algorithmiques. Le but étant toujours de gérer au mieux la circulation au sein de l'immeuble (éviter les attentes) ;
- l'interdiction de s'arrêter au premier et au deuxième étage de l'immeuble...(quel modèle intéressant...)

Toutes ces options sont données à titre d'exemple. Nous vous laissons la plus grande liberté dans les options que vous voudrez ajouter. Cependant, Vous devrez justifier d'un point de vue algorithmique vos améliorations.