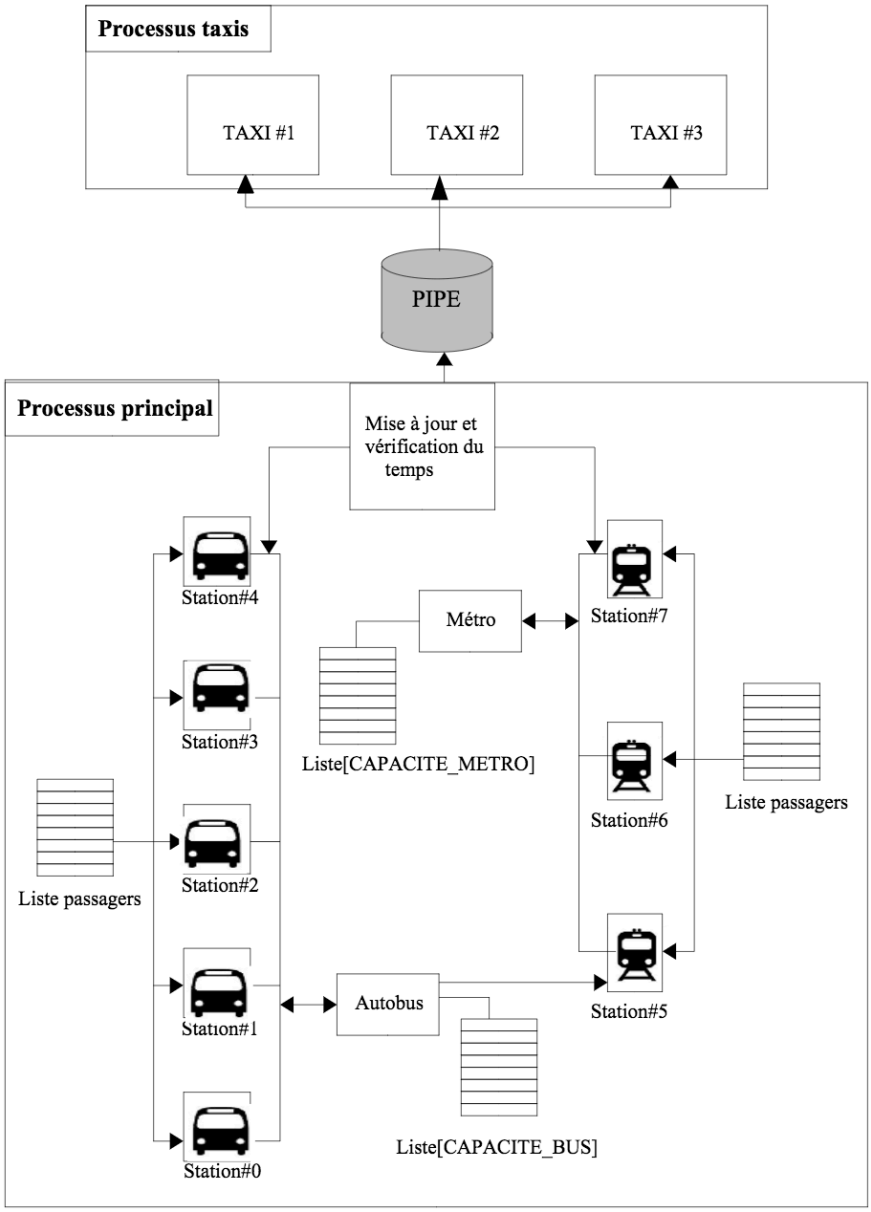
Simulation d'un système de transport en commun

Mohammed EL KHAIRA

2017 - 2018



Introduction

L'objectif de ce projet est de nous familiariser avec les appels système Unix/Linux concernant la communication et la synchronisation des processus et des threads. Il s'agit de simuler un circuit d'un système de transport en commun, composé d'un autobus, d'un métro et de trois taxis. Nous fournissons en plus de ce rapport le code C utilisé pour réaliser ce projet, ainsi que des jeux de données sur les passagers.

Description globale de notre approche

Pour ce projet, nous avons adopté les directives de l'énoncé. Nous avons un processus principal et un processus fils. Le processus principal crée trois threads : bus, metro, et verificateur. Le processus fils crée trois threads également : ce sont tous les trois des threads taxis.

Afin de pouvoir synchroniser tout cet ensemble, nous avons utilisé un mutex (**nb**), trois sémaphores (**semBus**, **semMetro** et **semVerif**), deux variables globales de types entier (**nbPassagersTotal** et **depensesTotal**), une variable globale de type pid_t (**pidFils**) et une structure sigaction (**action**) afin de pouvoir tirer profit de l'envoie de signaux entre le processus père et son fils.

Les variables globales viennent du fait qu'elles doivent être communes aux threads, le mutex permet d'éviter les modifications simultanés de ces variables, les sémaphores sont là pour mettre en place le rendez-vous bilatéral demandé et l'envoie du signal SIGUSR1 permet de synchroniser le père et le fils lors de la terminaison du programme (à l'aide de la routine **count**).

Avec cette approche, nous n'avons besoin que d'une **seule** structure centrale : une structure regroupant les files d'attentes de toutes les stations. C'est cette structure qui sera donnée en paramètre lors de la création des threads bus-metro-verificateur.
Les threads taxis quant à eux ne prennent que leur id en paramètre.

Enfin, nous avons utilisé quelques **constantes** :
MAXBUS, **MAXMETRO**, **NBSTATIONS_BUS**, **NBSTATIONS_METRO**, **NBTAXIS.**
Elles concernent le nombre de stations, la capacité max bus, la capacité max métro et le nombre de taxis. Elles sont définies dans **struct.h :**

Découpage de notre programme

Nous avons choisi de découper notre programme en 4 fichiers sources :
- **liste.c** : fonctions liées à la manipulation des listes.
- **chargement_fichier.c** : fonctions liées au chargement des données des passagers.
- **main.c** : fonction principale.
- **thread.c** : fonctions liées aux threads crées dans le main.

De plus, comme nous l'avons mentionné précédemment, il y a le fichier struct.h contenant la définition des constantes et des structures utilisées.

Enfin, plusieurs fichiers texte sont présents. Ils contiennent différents jeux de données sur les passagers.

Structures de données choisies

Les structures utilisées sont implémentées dans le fichier **struct.h**. Il y a :

- la **structure liste** : elle représente une liste de passagers. Elle nécessite la définition d'une **structure maillon** (contenant un seul passager).

- la **structure passager** : elle représente un passager. Elle contient les champs décrits dans l'énoncé, à savoir les entiers **id**, **stationDepart**, **stationArrivee**, **tempsEcoule**, **tempsMax**, et le booléen **transfert** qui renseigne sur le parcours du passager. En plus de ces champs, nous avons ajouté le champ **depenses**. Cet entier nous permet de pouvoir déterminer ce que chaque passager a dépensé pour son trajet (mais nous aurions pu nous en passer).

- la **structure listeAttente :** elle permet de stocker la liste de passagers en attente à chaque stations. La structure contient trois tableaux de liste :
  - **bus[NBSTATIONS_BUS]** :
    pour stocker les listes des passagers attendant le bus.
  - **metro[NBSTATIONS_METRO]** :
    pour stocker les listes des passagers attendant le métro à destination de la station 7.
  - **metroInverse[NBSTATIONS_METRO]** :
    pour stocker les listes des passagers attendant le métro à destination de la station 5.

Chaque case d'un tableau correspond à une liste de passagers en attente à une station.

Rôle des sémaphores :

Nous avons utilisés trois sémaphores : **semBus**, **semMetro**, **semVerif**. Ces sémaphores servent à synchroniser les threads **bus-metro-verificateur**.

En effet, nous devions faire en sorte que le thread verificateur attende que le bus et le métro aient fait un tour de boucle (débarquement, transfert puis embarquement des passagers) avant de s'exécuter. De même, les threads bus et metro doivent se suspendre le temps que verificateur fasse un tour de boucle (incrémentation du temps d'attente et transfert vers les taxis).

Pour cela, nous avons initialisé semBus et semMetro à 1 et semVerif à 0 : semBus, semMetro sont décrémentés une fois au début du cycle de bus, metro ; semVerif est décrémenté deux fois au début du cycle de verificateur. Parallèlement, semBus, semMetro sont incrémentés une fois chacun à la fin de verificateur ; semVerif est incrémenté une fois à la fin de bus et une fois à la fin de metro.

Nous avons donc mis en place un système de **rendez-vous bilatéral** : le thread verificateur a besoin que bus et metro soient en fin de cycle pour démarrer ; il en est de même pour ces derniers.