

**Rapport du devoir de système d’exploitation**

Alexandre KARAKAS - Marc CHOU

Ingénieur Informatique 1



Année 2017 - 2018

**Table des matières**

[1. Déroulé du projet 1](#_Toc1219151167)

[1.1. Préparation du projet 1](#_Toc1254600334)

[1.2. Processus principal 1](#_Toc240823536)

[1.3. Les threads autobus et métro 2](#_Toc917439799)

[1.4. Le thread vérificateur 3](#_Toc474863057)

[1.5. Le processus taxis 3](#_Toc407396744)

[2. Amélioration du projet 4](#_Toc140036798)

# 1. Déroulé du projet

## 1.1. Préparation du projet

Dans un premier temps, il a fallu préparer l’environnement de travail. Nous avons choisi d’utiliser l’IDE (*Integrated Development Environment*) *CLion* pour ses multiples fonctions pratiques. On peut citer son débogueur permettant de parcourir le code pas à pas, sa fonction complétion de mot et bien d’autres outils. Ensuite, nous avons construit un dépôt de version, sur la plateforme *GitHub*, pour pouvoir partager notre code plus facilement et retourner vers d’anciennes versions de code en cas de problème. Après cette préparation, nous avons relevé les informations importantes contenues dans le sujet.

## 1.2. Processus principal

Le projet a demandé l’utilisation de file FIFO. Nous avons décidé de reprendre le code d’une pile, dans notre cours de *Structures de données*. Il permet de manipuler plus facilement le contenu et répondre aux cahiers des charges. De plus, ce code permet de ne pas “sur-allouer” de l’espace mémoire mais, d’allouer l’espace nécessaire pour chaque file.

Parmi ses fonctions classiques, nous avons la fonction push qui permet de rajouter des passagers dans la file. Pop nous permet de supprimer une donnée de la liste.

Selon le cahier des charges, la structure d’un passager doit être la suivante :

|  |
| --- |
| **typedef struct** {  uint32\_t identification\_number; // Numero d'identification unique  uint8\_t station\_start; // Station de depart  uint8\_t station\_end; // Station d'arrivee  uint32\_t wait\_time\_past; // Temps d'attente ecoule  uint8\_t transfert; // Transfert entre le circuit de metro et d'autobus est requis  uint32\_t wait\_time\_maximum; // Temps d'attente maximal }passenger; |

*Structure d’un passager*

Lors de la récupération des données sur le fichier, nous avons pensé que le nombre de passager était inutile. Or, plus tard, nous avons remarqué qu’il avait son utilité pour la condition d’arrêt de certaines boucles while. Nous avons décidé de déclarer une variable globale pour récupérer la valeur et la décrémenter par la suite. Après cela, nous avons récupéré chaque passager avec une fonction pour l’inclure dans la bonne file d’attente.

Nous avons mis en place le tube nommé *communication.fifo* avec la fonction *mkfifo*. Avec *fork( ),* nous avons pu créer un processus taxi. Ensuite, nous avons envoyé en paramètre les files d’attente qui contenaient des passagers. On parcours ces files pour récupérer ou débarquer les passagers à chaque station.

|  |
| --- |
| **char** \*myfifo = "communication.fifo";  // Creation d'un tube nomme avec permission : READ, WRITE, EXECUTE/SEARCH by OWNER **if**(mkfifo((myfifo), S\_IRWXU) == -1) {  fprintf(stderr, "Erreur lors de la creation du tube.\n");  exit(EXIT\_FAILURE); } |

*Creation d’un tube nommé communication.fifo*

## 1.3. Les threads autobus et métro

Quelques problèmes ont été soulevés durant l’implémentation de ces threads.

Le processus nous a renvoyé un signal SIGSEGV. Ce problème est dû au fait d’accéder à un pointeur NULL durant la modification d’une file de passagers d’un autobus ou d’un métro. La fonction se chargeant de cela, brisait la chaine de la file. Elle ne changeait pas l’adresse du maillon suivant du maillon précédent. Ce qui impliquait que le maillon précédent conservait l’adresse du maillon supprimé.

Pour résoudre ce problème, nous avons repris la fonction *remove\_chain* implémentée en *Structures de données*. De plus, nous avons implémenté une fonction pour récupérer la position du maillon à supprimer, utile à la fonction précédemment mentionnée.

Lorsque nous avons exécuté notre programme, le processus s’est figé et a tourné en boucle. Après l’usage du débogueur, nous avons pu détecter et régler cette erreur qui s’est glissée dans une condition du thread métro.

|  |
| --- |
| **else if**((count\_station == 0) && (chain\_subway->data->transfert == 1)) |

*Condition incorrecte provoquant une boucle infini dans le while*

La condition vérifiait le transfert des passagers vers la queue de la station de bus.

|  |
| --- |
| **else if**((count\_station == MAX\_STATION\_BUS) && (chain\_subway->data->transfert == 1)) |

*Correction de la condition*

Mise à part ces problèmes, nous avons utilisé quatre sémaphores. Soit A, B, C, D, les quatre sémaphores que nous allons expliquer leur rôle et leur valeur initiale.

A est le sémaphore pour bloquer le cycle du thread bus. Il est initialisé à 1. Le thread enverra une requête post pour débloquer le sémaphore C. Il utilisera ensuite, la requête wait pour attendre la fin de l’exécution du cycle du vérificateur.

B est le sémaphore pour bloquer le cycle du thread metro. Il est initialisé à 1. Le thread enverra une requête post pour débloquer le sémaphore D. Il utilisera ensuite, la requête wait pour attendre la fin de l’exécution du cycle du vérificateur.

C et D sont les sémaphores pour bloquer le cycle du thread vérificateur. Il est initialisé à 0. Le thread commencera par utiliser la requête wait pour attendre la fin des exécutions des cycles du bus et du metro. A la fin, il utilisera la requête post pour débloquer les sémaphores A et B.

## 1.4. Le thread vérificateur

L’implémentation du tube de communication a été un peu compliquée. Nous nous sommes posés la question de savoir si nous pouvions passer une structure dans un pipe et si nous pouvions la récupérer. Pour répondre à ce problème, nous avons fait des recherches sur internet et tester les solutions proposées. Durant ces tests, un problème récurent est survenu :

|  |
| --- |
| \*\*\* stack smashing detected \*\*\*: <unknown> terminated |

*Message d’erreur*

Après l’utilisation de la fonction *printf* sur certaines parties du code, nous avons pu déterminer quelle fonction posait problème. La fonction *read* était la source.

|  |
| --- |
| read(fd, &passenger2, **sizeof**(passenger)); |

*Fonction read posant problème*

Cette fonction lit jusqu’à la taille d’une structure *passenger* d’octet depuis le descripteur de fichier *fd* dans le tampon pointé par *passenger2*. Or, lorsque nous avons écrit la fonction *write*, nous avons passé l’adresse d’un *passenger*. Donc, la taille d’octet à lire est incorrecte. La solution est la suivante :

|  |
| --- |
| read(fd, &passenger2, **sizeof**(int32\_t)); |

*Fonction read résolue*

## 1.5. Le processus taxis

Le principal problème est la mise en place de ce tube de communication. Il semblerait que la ré-ouverture en lecture par les taxis ne fonctionne pas ou bien, la récupération de la structure mène à une structure parfois vide ou aberrante. On peut aussi noter le fait que l’écriture n’arrive pas à écrire ou se bloque dans le thread vérificateur.

Après consultation avec un collègue, il semblerait qu’il ne faut pas ouvrir et fermer le pipe dans une boucle while. Suite à son conseil, nous avons placé l’ouverture et la fermeture du tube en dehors des boucles while. Cela a eu pour conséquence de régler le problème.

En revanche, un autre problème est survenu. La condition d’arrêt de la boucle while n’est pas remplie. Nous avons pensé au début qu’attribuer une variable globale permettrait d’arrêter la boucle mais, nous nous sommes rappelés que la duplication de processus dupliquait les variables globales. Ce qui signifie que le changement dans les threads du processus père n’affectent pas les threads du processus fils.

# 2. Amélioration du projet

Le cahier des charges demandait à ce que le thread vérificateur envoyait le passager à un taxi spécifique. Nous avons décidé qu’il serait plus pertinent que tous les taxis récupèrent le passager. Dès qu’un taxi est disponible, il récupère directement le passager. Il n’y a donc pas de choix de taxi.

Pour étendre le simulateur à plusieurs bus circulant dans le même sens, il faut déclarer une constante contenant le nombre de bus souhaité. Ensuite, il faudra allouer dynamiquement un tableau de pthread\_t pour obtenir plusieurs identifiants de thread. Après cela, une boucle for sera utilisée pour créer plusieurs thread.

La fonction associée au thread devra contenir un semaphore et un mutex en plus pour utiliser le rendez-vous bilatéral. On déclarera un compteur en variable général en plus, initialisé à 0. Il faudra :

- verrouiller avec le mutex à la fin du while,

- incrémenter le compteur,

- vérifier si le compteur est égale au nombre de bus,

- utiliser la fonction post pour l’arrivé de tous les threads,

- déverrouiller le mutex

- utiliser la fonction wait pour attendre tous les threads

- utiliser enfin la fonction post pour libérer.

Le sémaphore sera initialisé à 0. A noter qu’il faudra utiliser la fonction pthread\_join() à la fin du fork() et désallouer ce qui a été alloué.

L’ajout de bus n’est pas nécessaire après une certaine limite. Il n’existe pas beaucoup de stations. En revanche, si le nombre de station augmente alors, il serait intéressant d’ajouter des autobus. D’ailleurs, il est aussi possible d’ajouter des métros en plus.

D’autre part, il est possible d’ajouter une pile contenant les passagers arrivés par ordre d’arrivé dans le temps. Cette pile permet de vérifier avec exactitude que chaque passager soit arrivé à la bonne station. Par contre, le fait d’implémenter cette pile consomme une partie de la mémoire.

Il faudra noter que la manière dont sont implémentées les files d’attentes n’est pas pertinente pour la durabilité du projet. En effet, imaginons que l’on souhaite ajouter stations avec d’autres plateformes de changements. Il est plus difficile de récupérer la station dans un tableau de file d’attente.