UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

RAPPORT DE PROJET : LE PEAGE AUTOROUTIER

Albert ROYER - Nicolas MARCELIN

INFO01

SY40 – A22

utbm
université de technologie
Belfort-Montbéliard

Table des matières

INTRODUCTION

Ce projet a pour but de mobiliser l'ensemble des connaissances acquises durant le semestre A22 de l'UV SY40.

Pour cela, nous devrons modéliser et implémenter le fonctionnement d'un péage autoroutier : On dispose de 5 classes de véhicules différentes, chaque véhicule peut payer au péage en espèce ou par carte ou en passant par l'abonnement télépéage (il existe donc deux types de péages, classique et télépéage). Il existe également une voie de covoiturage, utilisable seulement en heure de pointe ; cette voie est réservée à certains automobilistes (tous les véhicules avec deux occupants ou plus, les taxis, et les véhicules avec la vignette « Crit'Air 0 »).

Le code est consultable sur GitHub : https://github.com/Rarynn/PEAGE-Autoroutier.

CHOIX

Dans un souci de réalisme, nous avons pris plusieurs dispositions concernant le projet :

Nous avons associé à chaque véhicule une plaque d'immatriculation, ne connaissant pas le rapport exact de véhicule ayant la vignette « Crit'Air », nous avons mis un rapport de 0.5 pour ce dernier et un rapport de 0.4 l'abonnement télépéage.

Nous avons également interdit la fermeture de tous les péages à l'exécution du programme, effectivement, ce cas de figure n'arrivera jamais en réalité.

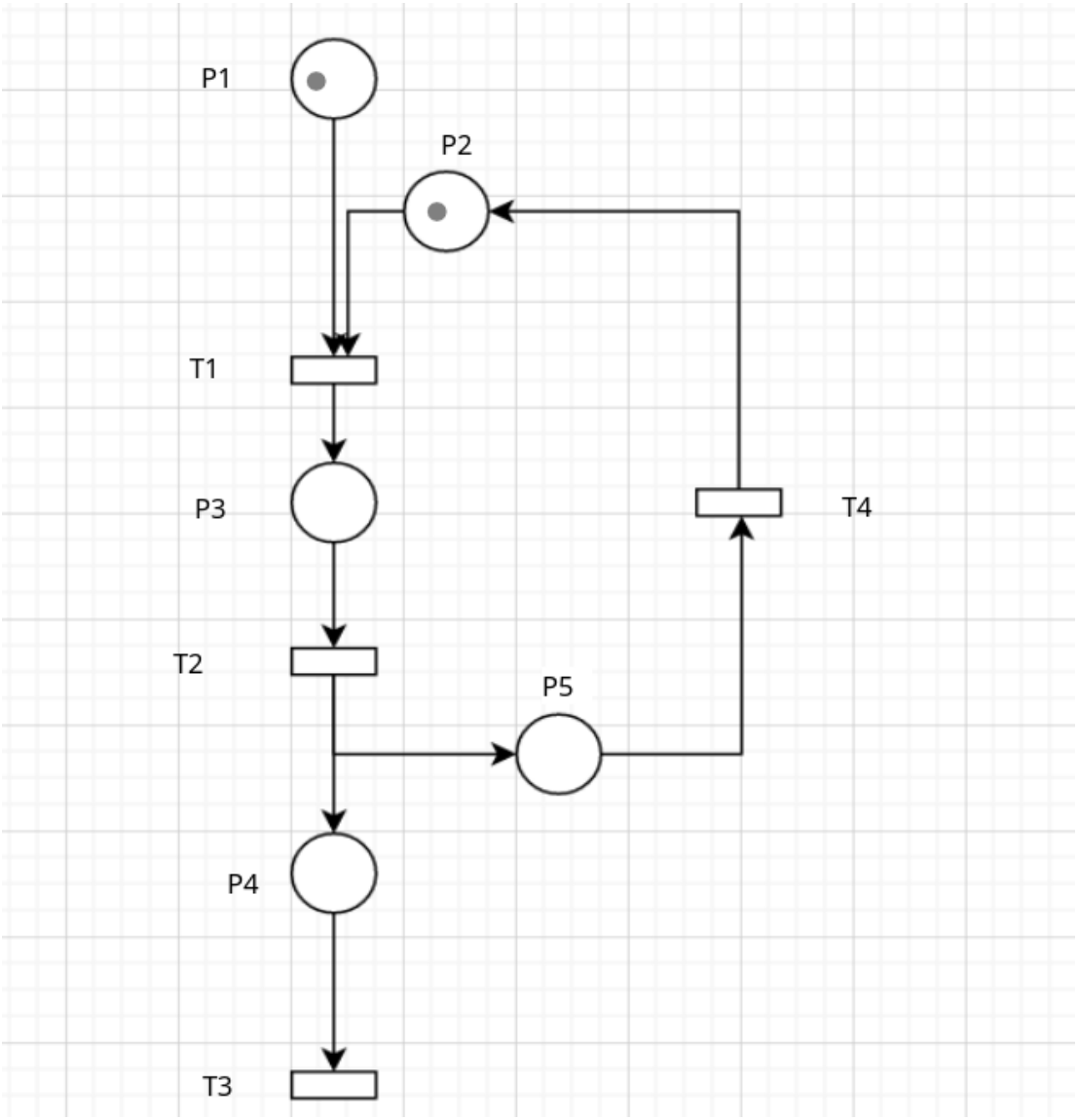
Concernant les voies, ¼ d'entres elles sont réservées au télépéage, la voie 0 servira, en période d'heures de pointes, de voie spéciale covoiturage.

Nous attendons la bonne terminaison des threads péages, ce qui peut être plus ou moins long dépendant la taille de **NB_PDP**, nous aurions pu utiliser d'autres fonctions tels que **pthread_cancel**, mais cela n'aurait pas été très propre au niveau de l'utilisation des ressources.

Pour éviter les erreurs de cast lors de la compilation, nous avons choisi d'utiliser PRAGMAS, une directive préprocesseur, cela nous a permis de découvrir de nouvelle macro non standard tout en ayant une compilation plus propre.

MODELISATION

Avant d'écrire le code, il nous a semblé important de modéliser le problème, afin de voir plus facilement les interactions entre un péage (une station) et des véhicules. Nous avons donc modélisé le problème sous forme d'un Réseau de Petri :



T1 : la voiture entre au péage et le réveille

T2 : la voiture paie le ticket

T3 : la voiture sort du péage

T4 : le péage se rendort

Ce réseau de Petri nous permet donc de simuler le partage de ressources critiques.

APPROCHE

Pour représenter les différents acteurs du système, nous avions la possibilité de choisir entre processus ou thread : pour simplifier le problème et dans un souci d'efficacité nous avons choisi les threads.

Le code sera donc constitué de **NB_PDP** threads représentants les péages, et de *nb_voiture* threads voitures. Nous utilisons également des moniteurs pour gérer tous ces threads différents.

SYNCHRONISATION & COMMUNICATION

Pour gérer l'aspect de la synchronisation, nous nous sommes basés sur les travaux effectués en CM et TP : le coiffeur endormi.

Effectivement, on peut faire un parallèle entre le coiffeur et un péage quelconque, et le client et la voiture. Ainsi, nous avons fait le choix d'utiliser des tableaux de variables de conditions (pthread_cond_t) et des tableaux de mutex (pthread_mutex_t) pour assurer la synchronisation entre threads voitures et threads péages.

La variable de condition « attente_voiture[pdp_id] » est utilisée pour signaler aux threads voiture de s'exécuter et de passer au péage correspondant. Dans la fonction Voiture, le thread voiture appelle la fonction pthread_cond_wait() pour attendre le signal du péage correspondant, qui sera envoyé grâce à la fonction pthread_cond_signal() dans la fonction Peage.

De manière similaire, la variable de condition « attente_peage[pdp_id] » est utilisée pour signaler aux threads péages de s'exécuter et de permettre à une voiture de passer. Dans la fonction Peage, le thread péage appelle donc la fonction pthread_cond_wait() pour attendre le signal d'une voiture en attente, qui sera envoyé grâce à la fonction pthread_cond_signal() dans la fonction Voiture.

Les mutex « mutex[pdp_id] » sont utilisés pour protéger l'accès à la variable partagée « nb_voiture_attente[pdp_id] » qui sert à compter le nombre de voitures en attente au péage correspondant.

Ainsi, seul un thread à la fois par péage peut accéder à cette variable et la modifier, ce qui permet de garantir la cohérence des donné. Néanmois il a était nécessaire de créer un autre mutex « sommemutex », afin de protéger la fonction « choix_pdp » qui accède directement à un tableau commun à chaque thread peu n'importe son péage. Nous avons donc été

dans l'obligation de penser la synchronisation des threads inter et intra péage.

DEBUG & TESTS

Concernant la fuite mémoire, nous avons utilisé l'outil de profilage VALGRIND, en utilisant la commande **«./exec main 1 10 »** nous obtenons :

```
==4294==
==4294== HEAP SUMMARY:
==4294==     in use at exit: 0 bytes in 0 blocks
==4294==   total heap usage: 28 allocs, 28 frees, 10,560 bytes allocated
==4294==
==4294== All heap blocks were freed -- no leaks are possible
==4294==
==4294== For lists of detected and suppressed errors, rerun with: -s
==4294== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Nous n'avons donc pas de fuite mémoire dans notre programme.

De plus, nous avons également utilisé différents flags dans notre makefile (bien qu'il soit rudimentaire) pour comprendre d'éventuelles erreurs.

AMELIORATIONS

Dans notre programme, le temps de fermeture des threads péages peut être plus ou moins long dépendant la taille de NB_PDP (le nombre de péages totaux), nous pourrions compter les voitures sorties, et rajouter une condition dans la fonction péage pour que celle-ci ne s'endorme pas si le nombre de voitures sorties égale le nombre de voiture générée, ce qui simplifierait la boucle qui attend la fermeture des threads dans le **main**.

Concernant la gestion dynamique du nombre de voies ouvertes en fonction du trafique, nous choisissons le nombre de voies qu'on ferme lors de l'exécution du programme, cependant il serait intéressant de pouvoir également choisir le nombre de voies totales (nous utilisons pour l'instant une constante définie en macro).

CONCLUSION

Ce projet a été plus complexe que les sujets de TP précédents en raison de la nécessité de trouver une solution adaptée à notre modèle. Nous avons exploré de nombreuses options pour faire communiquer et synchroniser nos threads.

Ensuite, faire fonctionner ensemble tous ces threads a été un autre défi important. Identifier les erreurs n'a pas été une tâche aisée : après quelques heures de réflexions, nous nous sommes rendu compte que notre aléatoire ne marchait pas à cause d'un problème d'initialisation de seed (srand était dans une boucle …).

Pour conclure, ce projet nous a permis d'approfondir nos connaissances sur la programmation système sous Linux, mais également de développer nos compétences concernant le makefile et la compilation.

Merci pour votre lecture.