PROJET INFO0601

Sacha KIERBEL
sacha.kierbel@etudiant.univ-reims.fr

Morvan LASSAUZAY
morvan.lassauzay@etudiant.univ-reims.fr

Intervenants :

Cyril RABAT ; Pierre DELISLE

Sommaire

INTRODUCTION

Dans le cadre des modules de Programmation Système et Programmation Multithread, il nous a été demandé de développer un simulateur de circulation routière comprenant un éditeur permettant de créer une carte constituée de routes et de différents éléments de signalisation tels que les feux et les stops. En interagissant avec cette carte affichée à l'aide de la bibliothèque ncurses, il est possible de créer des voitures capable de se déplacer sur les routes en suivant les règles de circulation définies par les éléments de signalisation. Chaque voiture correspondant à un thread, il a été indispensable d'utiliser la bibliothèque pthread qui nous a également permis de gérer le changement de couleur des feux et l'envoi de messages entre les différentes applications. En effet l'application globale est décomposée en plusieurs applications : le coordinateur et les clients d'affichage. Afin de communiquer entre ces différentes entités, il nous a fallu utiliser une file de message, un tableau de sémaphore et un segment de mémoire de mémoire partagée.

Afin de bien cerner le fonctionnement de ce simulateur, nous verrons dans un premier temps le rôle et la structure des différents éléments qui le constituent pour ensuite s'attaquer aux IPC utilisés et leur utilité et enfin terminer sur l'utilisation des threads dans notre application.

STRUCTURE DU SIMULATEUR

La carte

La carte est l'élément de base du simulateur car ce sont les éléments qui la constituent qui vont définir les actions à entreprendre. Elle est composée d'un ensemble de zones, chaque zone étant elle-même composée d'un ensemble de cases.

Une case est définie par une table de routage et un type qui peut être une direction (haut, bas, droite, gauche), un feu (vert ou rouge), un stop, ou encore un croisement. La table de routage n'est utilisée que dans le cas où la case est de type croisement ou feu. Lorsque la case est de type croisement, la table de routage permet d'indiquer une direction de sortie selon une direction d'arrivée. Lorsque la case est de type feu, la table de routage sert à indiquer à quel ensemble appartient le feu ainsi que son numéro de passage au vert (1, 2, 3 ou 4). Dans notre simulateur tous les feux possédant le même numéro passent au vert en même temps et cela indépendamment de l'ensemble auquel ils appartiennent.

La concaténation des différentes zones formées chacune d'une matrice de cases permet d'obtenir une carte globale correspondant à une grande grille de cases dont les dimensions peuvent varier selon les valeurs choisies lors de sa création. Lors de l'exécution du simulateur ce sont les clients d'affichage qui ont pour rôle d'afficher la carte intégralement ou en partie.

Les clients d'affichage

Peu importe les dimensions de la carte celle-ci est décomposée par le simulateur en quatre zones d'affichage. Un client d'affichage permet d'afficher une des ces zones et d'interagir avec en cliquant avec la souris. Une zone d'affichage ne peut pas être affichée par plusieurs clients en même temps mais si un client est fermé alors il est possible d'ouvrir un nouveau client afin d'afficher à nouveau la zone.

En cliquant sur une case de type direction ou croisement un utilisateur à la possibilité de créer une voiture. Si l'utilisateur clique ensuite sur cette voiture il la supprime. Une fois créée la voiture se déplace sur la carte en suivant les routes et les règles de circulation. Quand la voiture arrive au bord d'un client d'affichage elle continue son chemin sur le client affichant la zone voisine si elle existe, si ce n'est pas le cas alors elle est détruite.

Pour assurer son bon fonctionnement un client d'affichage a recours à l'utilisation de plusieurs threads : un pour permettre le changement de couleur des feux, un pour assurer le transfert de véhicule entre clients et l'envoi d'informations au coordinateur, et un thread pour chaque voiture qui circulera sur ce client. Nous détaillerons leur fonctionnement par la suite.

Le coordinateur

Le coordinateur peut être vu comme la tour de contrôle de l'application. Il doit être lancé en premier car il permet l'initialisation des éléments nécessaires au bon fonctionnement général du simulateur. En effet lors de son exécution le coordinateur commence par récupérer la carte dans le fichier binaire qui la contient. Il peut ensuite

créer et initialiser les différents outils de communication entre processus (IPC). Le coordinateur a également pour rôles d'afficher au cours du temps l'ensemble des informations concernant la simulation. Pour finir il doit ordonner de manière centralisée le changement de couleur des feux.

Pour parvenir à tout cela le coordinateur a lui aussi recours à l'utilisation des threads. Il emploie ainsi un thread pour ordonner le changement de couleur des feux et un deuxième thread pour récupérer et afficher l'ensemble des informations concernant la simulation.

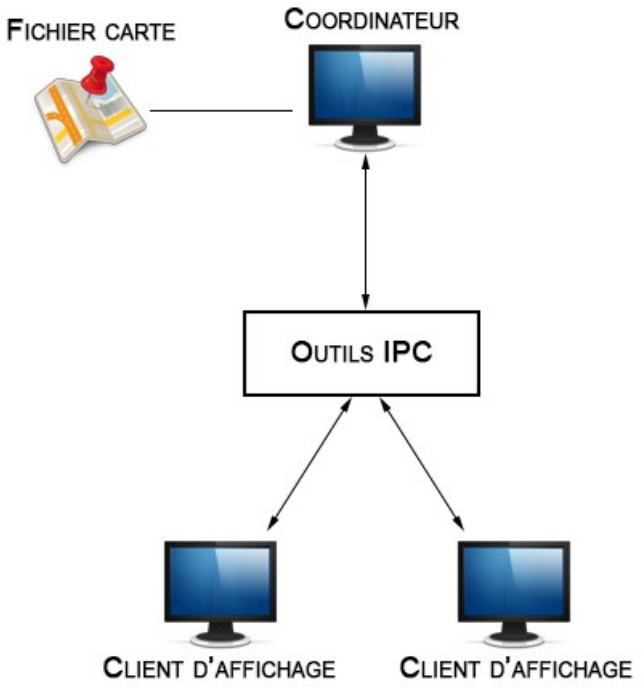

Figure 1: Structure générale de l'application

LES IPC

Le segment de mémoire partagée

Le segment de mémoire partagée permet de donner accès à la carte et aux informations la concernant à tous les processus de l'application. Après l'avoir créé, le coordinateur y place d'abord le nombre de colonnes de zones que contient la carte suivi du nombre de lignes de zones, puis y place ensuite toutes les zones une par une jusqu'à ce que l'intégralité de la carte s'y trouve.

Après s'être attachés au segment, les clients peuvent de leur coté récupérer les dimensions de la carte afin de pouvoir calculer les dimensions de la zone à afficher, puis selon leur numéro de client ils récupèrent un pointeur sur chacune des zones comprises dans leur zone d'affichage.

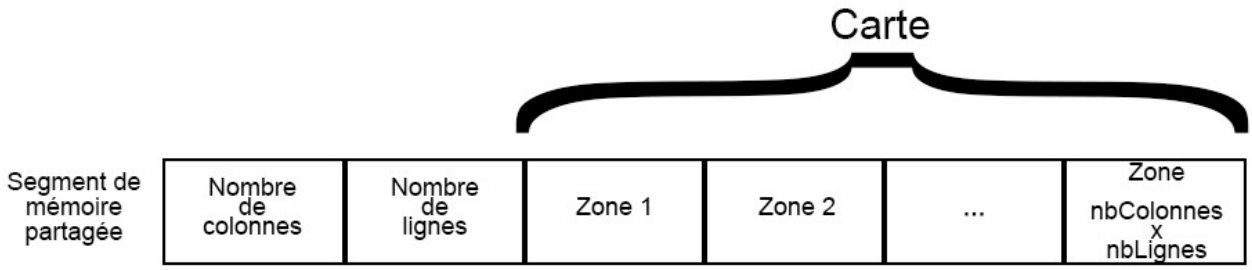

Figure 2 : Segment de mémoire partagée

Le tableau de sémaphores

Le tableau de sémaphore de notre simulateur va permettre la gestion de plusieurs éléments lors de la simulation.

Les quatre premiers sémaphores qu'il contient ont pour rôle d'accorder ou non l'acquisition d'une zone d'affichage à un client. Ces quatre sémaphores correspondant aux quatre zones d'affichage sont initialisés à 1 dans un premier temps. Ensuite lorsqu'un client acquiert une zone d'affichage le sémaphore correspondant à cette zone est passé à 0 rendant alors impossible une future décrémentation jusqu'à la mort du client qui fera repasser la valeur du sémaphore à 1.

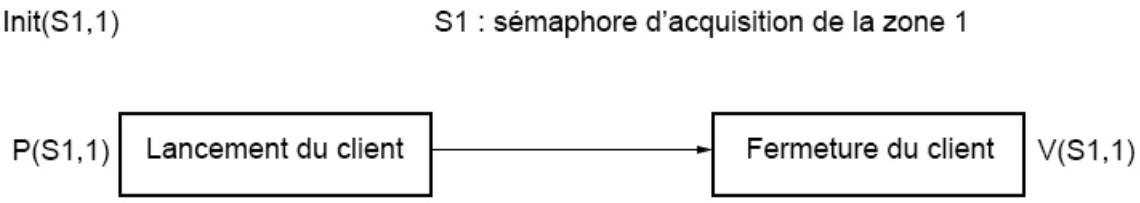

Figure 3 : Exécution d'un client d'affichage pour la zone 1

Le cinquième sémaphore de notre tableau sert quant à lui à contrôler le nombre de voitures présentes sur la totalité de la carte. Pour cela le coordinateur l'initialise au nombre maximum de voiture qu'il souhaite accepter sur la carte, puis pour chaque voiture créée le sémaphore est décrémenté de 1. Lorsqu'une voiture est supprimée le sémaphore est bien entendu incrémenté de 1.

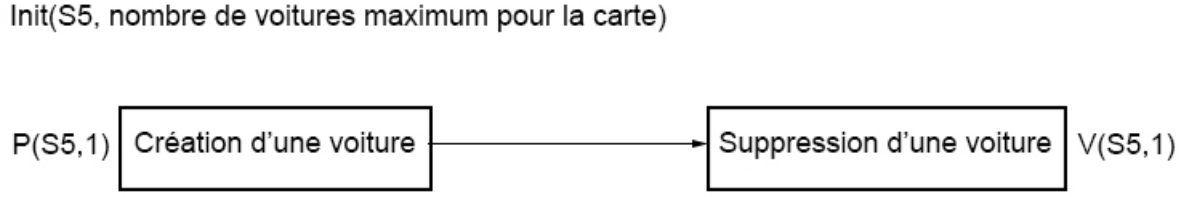

Figure 4 : Création et suppression de voitures

Les deux derniers sémaphores de notre tableau sont utilisés pour le changement de couleur des feux. Le premier peut prendre les valeurs 1, 2, 3 ou 4 pour indiquer le numéro correspondant aux feux vert (les feux possédant ce numéro passeront au vert). Le 2ème est d'abord initialisé à une valeur positive. Ensuite lors d'un changement de couleur de feux le coordinateur passe ce sémaphore à 0. Les clients qui étaient en attente du passage à 0 de ce sémaphore sont alors avertis du changement de couleur et effectuent l'opération. Dès qu'un client reçoit le signal de changement de couleur de feux il incrémente le sémaphore pour permettre de réaliser à nouveau le processus par la suite.

La file de messages

La file de messages est utilisée pour permettre le transfert de véhicule d'un client à un autre ainsi que pour l'envoi d'informations des clients au coordinateur. Pour ces deux actions la structure de message utilisée est la même. Cette structure comprend un type permettant d'indiquer qui est le destinataire du message ainsi que divers autres éléments utilisés ou non selon le besoin. Parmi ces éléments on trouve deux entiers pour pouvoir indiquer les coordonnées d'une voiture, un autre entier pour pouvoir indiquer la vitesse d'une voiture, et un champs de type chaîne de caractères pouvant contenir un message.

La gestion des événements et des threads

La structure et l'utilité des éléments composant notre simulateur ayant été définie, voyons désormais comment sont effectuées les différentes opérations par l'ensemble des threads constituants le programme en lui-même.

La gestion des voitures

Comme nous l'avons vu précédemment chaque voiture est en réalité représentée par un thread. Ces threads (que l'on appellera threads voiture par la suite) vont donc être en charge de réaliser les déplacements des voitures cela engendrant des accès concurrents à la carte. En effet afin d'effectuer un déplacement, un thread voiture a besoin de connaître et de modifier l'état de sa case (celle sur laquelle se trouve sa voiture) ainsi que celui d'une ou deux cases voisines. Cela ne pose pas de problème s'il s'agit de simple case direction et si aucune voiture ne s'y trouve, mais dès lors que l'on a affaire à une case de type feu ou une case sur laquelle se trouve une voiture alors cela devient problématique. Effectivement dans ces deux derniers cas le thread voiture a besoin de s'assurer de l'état réel de la case qu'il est entrain de consulter ou modifier, il risquerait sinon par exemple de croire qu'un feu est vert alors que celui-ci est passé au rouge et il continuerait alors son chemin malgré le feu rouge. Pour pare à cela le thread voiture doit être sûr qu'il est le seul à avoir accès à cette case lors de certaines opérations. Pour s'en assurer nous avons choisi de mettre en place une structure de grille couvrant l'intégralité de la carte où chaque case de la grille correspond à une case de la carte et dispose d'un mutex et d'une variable de condition permettant de définir l'accès aussi bien à la case de la grille qu'à celle de la carte les deux étant confondues. Les cases de la grille disposent également d'un entier servant à indiquer si une voiture se trouve sur la case et d'un pointeur vers le thread de cette voiture si elle existe. Maintenant que cette structure de grille est mise en place, les threads voitures disposent de l'ensemble des informations dont ils ont besoin pour effectuer leurs déplacements et sont capables de s'assurer de la

validité de ces informations.

Juste avant nous avons spécifié qu'une case de la grille possédait une variable de condition, mais quel est son but ? Cette variable de condition nous permet de mettre le thread voiture en attente lorsque celui-ci est bloqué à un feu rouge ou derrière une autre voiture. Cela nous permet de limiter l'exécution inutile des threads voitures s'ils ne peuvent pas avancer. Lors de la mise en place de ce procédé, nous nous sommes interrogés sur le gain d'utilisation du CPU engendré par l'attente des threads par rapport au coup d'utilisation du CPU engendré par les actions nécessaires à cette attente combiné à la quantité de ressources que cela nécessite. N'ayant pas les moyen de répondre à cette interrogation nous avons choisi d'utiliser les procédés vus en cours et d'empêcher au maximum toute exécution inutile des threads.

Un autre point qui nous a demandé grande réflexion a été la gestion de l'affichage. Nous avons tout d'abord songé à dédier un thread complet à l'affichage mais la seule opération que ce thread était en mesure d'effectuer était la mise à jour de la fenêtre ncurses une fois les différents éléments à afficher définis par les threads voiture. Cela aurait de plus nécessité différentes opérations supplémentaires entre les threads voitures et le thread d'affichage. Nous avons donc choisi d'inclure directement la mise à jour de l'affichage dans la méthode de déplacement des threads voitures, cela nous semblant moins coûteux en terme d'exécution que l'utilisation d'un thread dédié à l'affichage.

La dernière chose qu'il nous a fallu mettre en place au niveau des threads voitures a été leur suppression. En effet à tout moment l'utilisateur peut choisir de supprimer une voiture ou de fermer le client d'affichage et donc de supprimer toutes les voitures se trouvant sur ce client. Pour éviter que la suppression d'une voiture entraîne un état indésirable pour l'application il a fallu prendre en compte tous les points d'annulation possibles des threads voitures. Certains de ces points comme les attentes sur une variable de condition son implicites, d'autres doivent être définis afin de permettre la suppression des voitures au moment voulu. Par exemple nous avons choisi de définir un point d'annulation avant le déplacement de la voiture, une voiture ne devant pas pouvoir se déplacer si une demande de suppression a déjà été effectuée sur celle-ci. Pour gérer ces points d'annulation nous avons utilisé les procédures de nettoyage *pthread_cleanup_push* et *pthread_cleanup_pop*, celles-ci nous permettant de définir les actions à effectuer avant l'arrêt d'un thread voiture lorsqu'une demande de suppression a été ordonnée sur ce thread.

Détailler ici l'ensemble des opérations sur les threads voiture et mutex de case serait trop long et certainement peu clair, c'est pourquoi nous invitons le lecteur à regarder le code du programme client que nous avons essayé de commenter au mieux, afin de réellement visualiser l'ensemble des opérations effectuées lors des déplacements.

La gestion des feux

Nous avons déjà vu auparavant que le contrôle du changement de couleur des feux faisait partie de la tâche du coordinateur. Pour ce faire, ce dernier utilise un thread entièrement dédié à cette action. Ce thread se contente d'effectuer les actions sur les sémaphores que nous avons déjà décrites dans la partie sur le tableau de sémaphores de notre simulateur.

Du coté des clients un thread particulier est également consacré au changement de couleur des feux. Ce thread enregistre dans un premier temps la position des feux sur la carte, puis il se met en attente sur le sémaphore donnant le signal de changement de couleur des feux comme nous l'avons également décrit dans la partie sur le tableau de

sémaphore. Une fois le signal reçu le thread examine les feux un par un et modifie leur couleur si besoin. Pour assurer la concordance entre les nouvelles couleurs des feux et les couleurs perçues par les voitures et l'utilisateur, cette opération nécessite de récupérer au préalable les verrous sur tous les mutex des cases de type feu ainsi que le mutex d'affichage. De cette manière l'opération de changement de couleur des feux peut s'effectuer de façon atomique au vu des autres éléments du programme et ainsi assurer à tout instant la validité des informations récupérées par ces autres éléments.

La gestion des messages et des transferts de voiture

Que ce soit le coordinateur ou les clients d'affichage, chacun dispose d'un thread lui permettant de récupérer les messages qui lui sont envoyés. Le type du message permet de désigner à qui le message est adressé. Ainsi les messages de type 1 sont destinés au coordinateur, ceux de type 2 sont destinés au premier client, et ainsi de suite. L'envoi d'un message peut quant à lui être effectué à n'importe quel endroit dans le programme.

Lors d'un transfert de véhicule un message contenant l'ensemble des informations sur la voiture à transférer est envoyé depuis la méthode *dep_out_client* d'un premier client, puis le thread relatif à la voiture sur ce client est détruit. Le client qui reçoit le message récupère alors ces informations afin de les passer en paramètre à un nouveau thread correspondant à la voiture. La création de ce nouveau thread peut engendrer un conflit si elle se produit en même temps que la création d'une nouvelle voiture suite à un clic de l'utilisateur sur la carte. Nous utilisons donc un mutex protégeant la création de véhicule de manière à pouvoir effectuer avec sûreté les opérations pouvant entraîner un dysfonctionnement.

L'envoi au coordinateur d'informations sur l’exécution de la simulation s'effectue très simplement. Le thread envoyeur rempli le champ de type chaîne de caractères du message avec le texte qu'il souhaite transmettre. Le thread permettant au coordinateur de lire les messages qui lui sont destinés n'a alors qu'à ouvrir les messages et afficher le texte.

Conclusion

Grâce à la mise en place d'un segment de mémoire partagée, d'un tableau de sémaphore et d'une file de message, il nous a été possible de partager des données entre les applications constituants notre simulateur tout en assurant la synchronisation entre ces différentes applications. L'utilisation des threads a quant à elle permis d'effectuer l'ensemble des opérations du programme de manière parallèle afin d'assurer le bon fonctionnement et la rapidité d'exécution de l'application. Malgré tout nous aurions aimer pouvoir proposer un arrêt centralisé de l'application suite à la fermeture du coordinateur, une gestion plus approfondie des erreurs et une plus grande factorisation du code à l'aide de fonction afin d'améliorer la qualité de l'application et de son code.