Projet PPC : Modélisation d'un croisement routier

Introduction :

Le but du projet est de faire un programme multiprocessus qui permet de modéliser un trafic routier lors d'un croisement. L'objectif est de mettre en œuvre les connaissances acquises sur la programmation parallèle et les principales spécificités derrière ce type de programmation.

Conception et choix techniques :

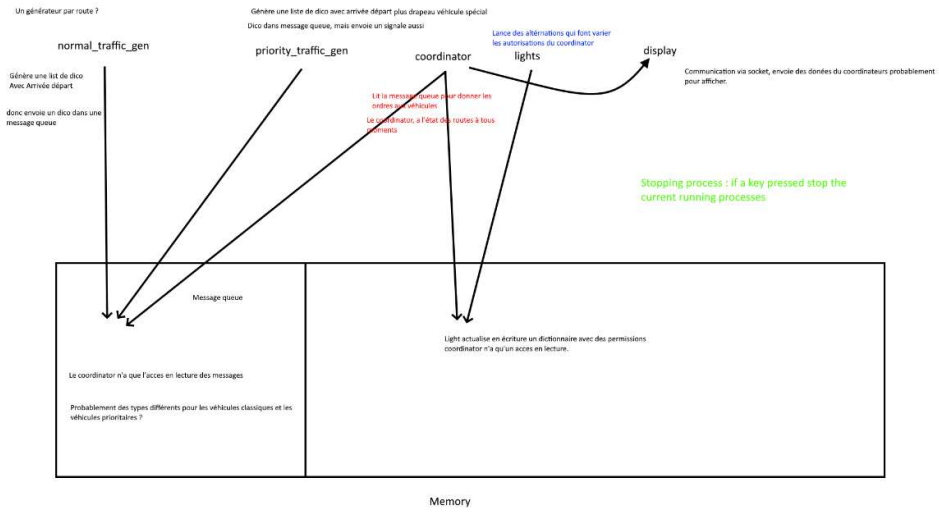

Figure 1 Première conception

Notre première visualisation du projet était celle donnée dans la figure 1, qui est relativement proche du résultat final. La différence majeure est les structures de données utilisées.

Au départ, nous avons commencé par concevoir les échanges au travers des messages queues, donc en créant le processus « normal_traffic_gen » et « coordinator » comme la gestion des véhicules prioritaires, et nous avons donc modifié la structure que les process envoient. En utilisant une chaîne de caractères « NS » pour désigner un véhicule qui part du nord pour aller au sud. Au lieu d'un dictionnaire compliqué.

Le traitement du coordinateur a été modifié, au départ nous avions commencé par un coordinateur par route, pour pouvoir faire avancer les deux routes vertes en parallèle. Mais cette idée a vite été abandonnée car overkill et ajoutait de la complexité aux communications interprocessus pour la gestion des priorités.

Ensuite la création du process « lights » qui permet de laisser passer les véhicules. C'est un des processus les plus simples, sur une boucle infinie, le processus modifie une variable écrite dans une shared memory que le coordinateur vient lire pour prendre ses décisions de lecture sur les messages queues.

Pour lancer et stopper les programmes de manière pas trop radicales, un process supplémentaire a été implémenté, avec uniquement une attente d'entrée, lorsque l'entrée est saisie elle permet de mettre fin à tous les autres processus de manière propre. De plus la création des shared memory y est faite, mais elles pourraient être créées dans les process qui y écrivent en premier.

Puis finalement le générateur de trafic prioritaire, qui fait la même chose que le générateur classique au détail près d'envoyer un signal au processus lights et d'utiliser un autre type, pour faire passer la route du véhicule au vert.

Puis finalement le display, qui reçoit les informations sous la forme d'une chaîne de caractères structurée. Le processus va parser le flux de données reçu dans le socket, les informations sont délimitées par des « _ » pour reconnaître le début et la fin d'une information. En cas d'information sur les feux l'information commence par « L » puis séparé par des virgules la première lettre de la direction des feux verts. Puis pour les véhicules, l'information commence par un « V », puis l'état du véhicule, « W » en attente d'un véhicule qui le laisse passer, « N » pour véhicule normal qui avance, « P » pour véhicule prioritaire qui avance, ensuite suivi de la direction de départ et la direction d'arrivée.

Architecture et Protocoles d'échange

Tout d'abord, certain échange interprocessus ont été supposé thread safe pour simplifier leur implémentation, surtout la shared memory.

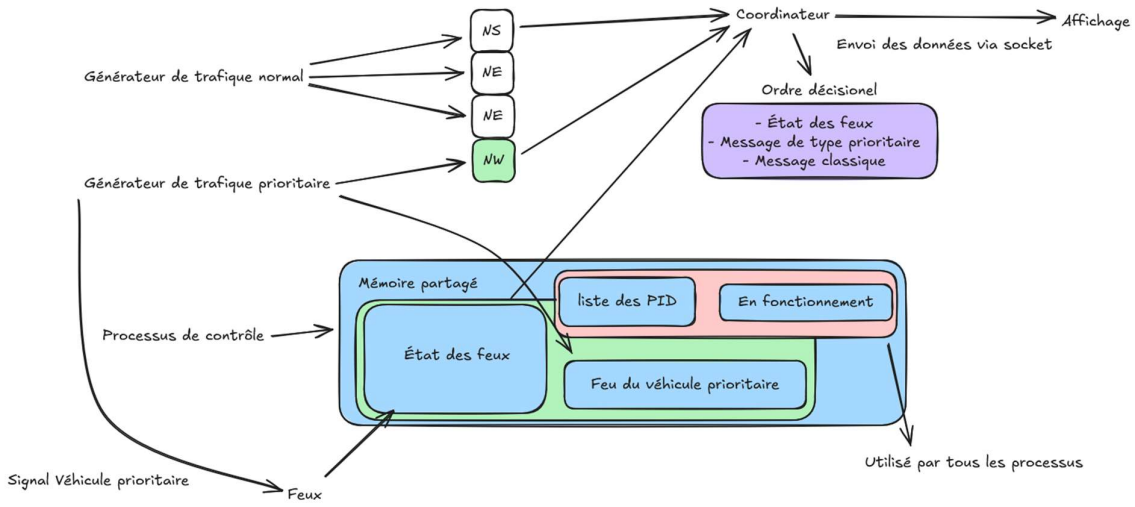

Figure 2 Architecture finale

Voici l'architecture finale des processus et leurs protocoles d'échange.

- La communication entre les générateurs et le coordinateur se fait via message queue, le message envoyé est comme expliqué précédemment sous forme de chaîne de caractères, « route de départ » « route d'arrivée », « NS » par exemple.
- Le générateur de trafic prioritaire va donc envoyer un signal au feu pour faire basculer les feux, et va écrire dans la mémoire partagée quel feu doit être modifié.

- Ensuite le processus de gestion des feux va modifier dans la mémoire partager les feux en permanence, et exceptionnellement lorsqu'un véhicule prioritaire est créé.
- Pour que les signaux soient envoyés et que les processus bouclent correctement, ils ont un accès en lecture dans la mémoire partagée à la liste des PID et une variable qui permet de stopper leur fonctionnement.
- Puis le coordinateur envoie les infos via un socket à l'affichage.

Algorithme important

Dans le cadre de ce projet, il n'y a pas vraiment d'algorithme à proprement parler, le code n'effectue pas de calcul. Le seul processus qui a une prise de décision complexe est le coordinateur. Voici l'idée derrière ce coordinateur :

Tant que le programme est actif :

Si les feux Nord/Sud sont verts :

Si un véhicule prioritaire attend :

Véhicule prioritaire passe

Sinon :

Si un véhicule vient du nord et ne doit pas céder la priorité :

Le véhicule passe

Sinon :

Le véhicule est stocké en mémoire

SI un véhicule vient du sud et ne doit pas céder la priorité :

Le véhicule passe

Sinon :

Le véhicule est stocké en mémoire

Puis la même chose avec les feux de la ligne Est/ouest. Ceci est l'idée générale de l'algorithme.

Tests :

Les deux communications interprocessus nécessitant beaucoup de test, sont : la communication entre les générateurs et le coordinateur pour s'assurer que le traitement des véhicules est bien le traitement voulu. Puis la communication entre le coordinateur et l'afficheur.

Pour ce faire, on teste en envoyant des types de véhicules précis, en donnant un ordre de véhicule écrit à la main au générateur, puis on vérifie les sorties du coordinateur. Si les sorties sont bonnes, on envoie donc les sorties à l'afficheur. Si les sorties sans traitements au bout du socket sont bien les sorties attendues, on passe ensuite aux tests de l'afficheur, en connaissant quel type de véhicule doit circuler, on peut voir dans l'afficheur si les véhicules qui circulent sont bien les véhicules souhaités.

Problèmes rencontrés :

Lors de la première implémentation avec un coordinateur par route, beaucoup de problèmes étaient présents, la gestion des priorités devait être faite entre les processus et ne simplifiait pas le problème, cette idée a donc été annulée.

La fermeture des messages queues et des shared memories, lorsque les programmes s'arrêtent, la fermeture doit être faite dans le bon ordre pour ne pas que des messages essaient d'être envoyés alors que la queue est fermée, de même pour l'accès à la shared memory.

Exécution du programme :

Le programme a un ordre particulier de lancement à cause de la création des messages queues, shared memory, sockets, dans l'ordre :

- Main.py, qui écrit dans la shared memory que les programmes doivent boucler
- Lights.py qui écrit l'état des feux.
- Normal_traffic_gen.py qui crée les messages queues et commence à les remplir
- Coordinator.py qui lit les messages queues et crée le socket
- Display.py qui va se connecter au socket et va faire que coordinator comme à écrire dans le socket (cependant coordinator fait quand même le traitement des véhicules même si le display n'est pas actif)
- Prio_traffic_gen.py si l'on veut ajouter des véhicules prioritaires dans nos queues.