Projet PPC - Crossroads

1. Objectifs du projet / Introduction

Un des objectifs de ce projet était de manipuler les outils de programmation parallèle et concurrente, notamment via les notions de processus, thread, mémoire partagée, et la synchronisation entre plusieurs entités.

Nous avons cherché à appliquer ces notions à travers un cas concret, qui est celui d'une simulation de carrefour routier : l'objectif de notre projet est donc de simuler un environnement de circulation avec différentes catégories de véhicules et des feux de signalisation bicolores, tout en mettant en place un système capable de réguler ces flux en fonction de règles prédéfinies.

Ainsi, notre code doit :

- Gérer l'apparition de véhicules normaux et prioritaires.
- Réguler le passage des véhicules en fonction des feux de circulation : si le feu est vert, une voiture classique peut passer. S'il est rouge, elle se stoppe si elle est encore positionnée avant ce feu.
- Gérer la priorité de certains véhicules en adaptant le cycle des feux : le feu de la route par laquelle un véhicule prioritaire arrive se met au vert, tandis que les trois autres passent au rouge. Ensuite, l'alternance ordinaire par binômes des feux est rétablie.
- Permettre une visualisation graphique permettant de suivre l'évolution du trafic en temps réel.

Ce projet nous a permis d'appliquer les concepts de programmation parallèle et concurrente dans un cadre réaliste et facilement visualisable, puisque la gestion d'un carrefour implique plusieurs entités indépendantes qui doivent communiquer et se synchroniser efficacement, ce qui illustre bien les défis rencontrés en programmation concurrente.

En outre, cette simulation permet de se confronter aux problématiques de partage de ressources (modification simultanée des états des feux), de communication entre processus (via des files de messages et de la mémoire partagée) et de gestion du temps réel (éviter des délais trop longs entre les changements de feux et les mouvements des véhicules).

2. Rendu visuel et utilisation du code

a) Interface graphique

L'interface graphique de notre projet permet de visualiser la simulation du carrefour routier en temps réel. Elle affiche :

- Les **routes et intersections** sur lesquelles les véhicules circulent.
- Les **feux de signalisation**, qui changent de couleur pour indiquer si les véhicules peuvent passer ou doivent s'arrêter.
- Les **véhicules**, représentés par des rectangles colorés, avec un code couleur permettant de différencier les types de trafic :
    - **Véhicules classiques** en bleu.
    - **Véhicules prioritaires** en rouge.

Les feux suivent une séquence prédéfinie de motifs de 5 secondes, mais ils peuvent être influencés par l'arrivée d'un véhicule prioritaire, ce qui interrompt le cycle normal pour lui donner la priorité, avant d'y revenir.

b) Interactivité, démarrage

L'utilisateur n'interagit pas directement avec l'interface graphique, qui fonctionne de manière autonome une fois lancée. Elle est seulement destinée à illustrer la simulation. Cependant, des paramètres peuvent être ajustés dans le code pour modifier le comportement du trafic (fréquence d'apparition des véhicules, temps de cycle des feux, gestion des priorités, etc.).

Les interactions principales avec le programme se font via le terminal, où l'exécution du script principal permet de démarrer la simulation. Un **README** accompagne le projet pour expliquer les commandes nécessaires à son exécution, mais un récapitulatif de lancement est fait ci-dessous.

- Ouvrir deux terminaux différents
- Pour chacun, se rendre dans le répertoire Crossroads du projet
- Lancer la simulation, run l'interface via le script *display.py* : *python3 display.py*
- De la même manière, run le *main.py* via *python3 main.py*

c) Technologies utilisées

Tkinter : utilisé pour l'interface graphique, permettant d'afficher les éléments du carrefour, les feux et les véhicules en temps réel.

Multiprocessing : utilisé pour gérer les différents composants du projet (simulation du trafic, gestion des feux, affichage) sous forme de processus indépendants.

Queus de messages (sysv_ipc) : employés pour la communication entre les processus, notamment entre la gestion des feux et la coordination du trafic.

Verrous et mémoire partagée : permettent d'assurer la synchronisation entre les processus et d'éviter les conflits d'accès aux ressources communes (états des feux, file d'attente des véhicules prioritaires).

3. Structure générale du projet

a) Architecture générale

Le projet est structuré en plusieurs modules distincts pour gérer les différentes fonctionnalités, illustrés dans la Figure 1 ci-dessous.

- **Gestion des feux de circulation (`lights.py`)** : Responsable du changement des feux en fonction des signaux et de la présence de véhicules prioritaires.
- **Gestion des véhicules normaux (`normal_traffic.py`)** : Génère un flux de véhicules réguliers et les ajoute aux files d'attente.
- **Gestion des véhicules prioritaires (`priority_traffic.py`)** : Simule l'arrivée de véhicules prioritaires et les insère dans une queue dédiée.
- **Coordination (`coordinator.py`)** : Gère la communication entre les feux, les véhicules et les files d'attente pour assurer un fonctionnement fluide. Communique par signaux avec *lights*, et par sockets avec *display*. Récupère les message queues des véhicules,et partage des données avec *lights*.
- **Affichage (`display.py`)** : Interface graphique permettant de visualiser l'état des feux et la circulation en temps réel.

Ces modules interagissent à l'aide de **queues de messages (`sysv_ipc.MessageQueue`), mémoire partagée (`multiprocessing.Manager().dict`), verrous(`multiprocessing.Lock`) et signaux (`signal`)**.

Via le main, le processus principal est créé et lance tous les processus secondaires (*light_process*, *normal_traffic_gen*, *priority_traffic_gen*, *coordinator*, *display*).

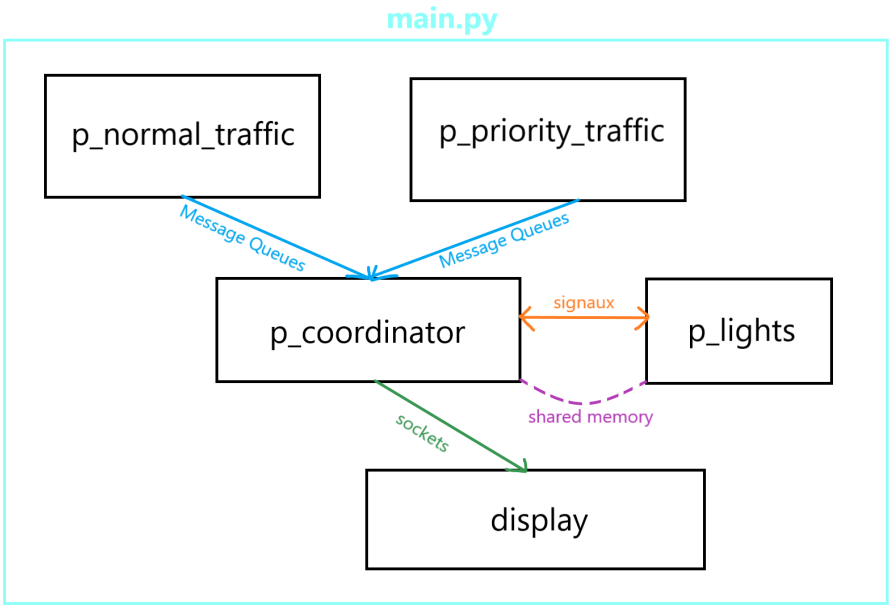

Figure 1 : schéma global de l'organisation des processus du projet

b) Relation parent-enfant entre processus

Dans notre programme, nous avons choisi d'orienter l'implémentation du code vers l'utilisation de processus créés et lancés séparément. Contrairement à une structure classique où un processus parent engendre des processus enfants et gère leur cycle de vie, ici, tous les processus sont démarrés indépendamment par le script principal (`main.py`).

Cette approche permet d'assurer une plus grande modularité : chaque module fonctionne de manière autonome et communique avec les autres via des mécanismes d'**IPC (Inter-Process Communication)** cités précédemment, tels que :

- **Les queues de messages (`sysv_ipc.MessageQueue`)** pour transmettre les événements liés aux véhicules et aux feux.
- **Les signaux (`signal`)** pour informer les feux qu'un véhicule prioritaire est en approche.
- **Une mémoire partagée (`multiprocessing.Manager().dict`)** permettant de stocker l'état actuel des feux en temps réel.

L'absence de relation parent-enfant stricte signifie que les processus ne sont pas directement liés les uns aux autres, ce qui leur permet de continuer à fonctionner même si l'un d'eux rencontre une erreur. Cependant, cette indépendance complexifie également la gestion de l'arrêt et de la synchronisation des processus, nécessitant un contrôle manuel via `terminate()` et `join()`.

En résumé, notre architecture repose sur une interaction **peer-to-peer** entre les processus, coordonnée principalement par le module `coordinator.py`, qui centralise les décisions et garantit la cohérence du système.

4. Choix de la structure de processus et gestion des processus

Le choix d'un modèle **multi-processus** permet d'optimiser l'exécution et la gestion des différents modules du programme. En séparant chaque tâche en processus distincts, nous bénéficions de plusieurs avantages :

- **Exploitation des cœurs du processeur :** En utilisant des processus séparés, nous pouvons tirer parti des multiples cœurs d'un processeur. Chaque processus s'exécute indépendamment, ce qui permet une gestion parallèle des tâches.
- **Meilleure séparation des tâches :** Chaque module est isolé dans un processus indépendant, ce qui rend le code plus clair, plus modulaire et plus facile à maintenir. Chaque processus s'occupe d'une tâche bien spécifique, et la communication entre les modules est facilitée par les mécanismes IPC cités plus haut.
- **Résilience et robustesse :** Si un problème survient dans un module (par exemple, un bug dans le gestionnaire de feux), il n'affecte pas directement les autres modules. Cette isolation minimise le risque de blocage du programme entier, permettant aux autres processus de continuer à fonctionner sans interruption.

Cependant, l'utilisation de processus multiples présente aussi des inconvénients :

- **Complexité accrue :** La gestion des processus, des queues de messages, et des mécanismes de synchronisation nécessite une gestion fine pour éviter les conflits. Le débogage est également plus complexe car les erreurs peuvent être isolées dans un processus particulier, rendant leur suivi plus difficile, ce qui nous a d'ailleurs pris beaucoup de temps à régler lorsque nous rencontrions des problèmes de ce type.
- **Synchronisation :** Les processus doivent être soigneusement synchronisés pour éviter des accès concurrents indésirables à des ressources partagées (comme l'état des feux de circulation). Une mauvaise synchronisation peut entraîner des erreurs difficiles à traquer.

5. Détail par module

a) Module de gestion des feux de circulation

Le **module de gestion des feux** contrôle les états des feux en alternant entre feu vert et rouge, avec une interruption en cas de véhicule prioritaire. Les signaux Unix (SIGUSR1, SIGUSR2, SIGTERM, SIGINT) permettent de réajuster immédiatement les feux, qui sont ensuite réinitialisés à leur cycle normal, le tout en utilisant des verrous pour éviter les conflits d'accès aux ressources partagées.s (`multiprocessing.Lock()`).

b) Modules de génération des véhicules

Les **modules de génération des véhicules** sont chargés de générer les véhicules qui circulent et de déterminer leur priorité. Les véhicules sont ajoutés à une file d'attente, et leur priorité est déterminée en fonction de leur attribut “priority” (True, ou False). Les véhicules sont traités dans l'ordre où ils arrivent, mais un véhicule prioritaire peut entraîner un changement immédiat du cycle des feux de circulation. Ce module interagit avec la **file de messages** (queues) pour envoyer des informations sur les véhicules et leur priorité au module des feux.

c) Module de coordination

Le **module de coordination** joue un rôle central en assurant la communication et la gestion synchronisée entre les différents modules (gestion des feux, gestion des véhicules, etc.). Ce module coordonne les décisions prises par le module de gestion des feux en fonction des messages reçus du module de gestion des véhicules. Il agit également en fonction des **signaux Unix** envoyés pour gérer les véhicules prioritaires. Par exemple, dès qu'un véhicule prioritaire est détecté, ce module reçoit l'information, interrompt le cycle des feux pour donner le passage, puis coordonne la réinitialisation des feux une fois le véhicule passé. Le module de coordination gère également l'**interface entre les processus** via des queues de messages, des shared memories, ou un serveur de sockets, et permet une coordination fluide du passage des véhicules dans l'ensemble du système.

d) Module d'affichage (GUI)

Le **module d'affichage (GUI)** est responsable de la visualisation de l'état des feux de circulation et des véhicules en temps réel. Il met à jour l'interface graphique en fonction des changements dans l'état des feux et de l'arrivée des véhicules. Cette interface fournit des informations claires aux utilisateurs sur l'état actuel des feux (rouge/vert) pour chaque direction et peut également afficher la présence de véhicules (prioritaires ou non) circulant sur les différentes routes. Le **rafraîchissement en temps réel** est assuré par l'interaction avec les autres modules via des queues de messages, ce qui permet de mettre à jour l'affichage dès qu'un changement important survient, comme un passage de véhicule prioritaire ou un changement de feu.

7. Problèmes rencontrés

Difficultés liées à la synchronisation :
La gestion des ressources partagées a posé des défis, notamment l'accès concurrent au dictionnaire `light_dict` qui contient l'état des feux de circulation. Des verrous (`multiprocessing.Lock()`) ont été utilisés pour éviter les écritures concurrentes, mais la gestion des verrous a généré des conflits de synchronisation. Les queues de messages ont également présenté des difficultés : si un processus envoie trop de messages sans les traiter, cela peut entraîner des blocages. De plus, la gestion des véhicules prioritaires a introduit une latence due à des boucles d'attente active.

Problèmes de temps réel et de performance :
Le changement des feux repose sur des intervalles de 5 secondes, mais cela entre parfois en conflit avec les véhicules prioritaires. La latence des signaux (comme SIGUSR1) permet de réagir rapidement aux véhicules prioritaires, mais peut causer des incohérences dans la synchronisation des feux. Nous avons aussi eu du mal à implémenter les premiers signaux par inexpérience.

Problèmes de gestion de processus :
La gestion des processus parallèles a soulevé des problèmes de synchronisation entre les différents modules (feux, véhicules, et coordination). Nous avons dû nous assurer que les processus n'entrent pas en conflit ou ne se bloquent pas.

Erreurs de communication inter-processus, et IPC :
Des erreurs telles que l'EOFError ont surgi lors de la gestion des queues de messages, notamment lorsque des queues étaient fermées de manière inattendue. Il a fallu gérer un peu plus rigoureusement l'ouverture et la fermeture des queues.

Difficultés liées à l'interface graphique
Des problèmes techniques ont surtout été rencontrés lors du chargement des données, la récupération avec une syntaxe correcte des sockets, et les liens à send_display à faire

8. Solutions apportées aux problèmes

Pour résoudre les problèmes de synchronisation, plusieurs solutions ont été mises en place. Nous avons utilisé des **verrous** (`multiprocessing.Lock()`) pour protéger l'accès aux ressources partagées (comme `light_dict`). Cela a permis de garantir que seule une tâche puisse accéder à ces ressources à un instant donné, évitant ainsi les conflits d'écriture. Pour la gestion des messages dans les queues, nous avons implémenté une logique de gestion d'attente active qui permet de réduire l'impact de la mise en attente des processus sans pour autant bloquer leur exécution.

De manière globale, il aura fallu créer beaucoup de print de valeurs et de variables, pour identifier les problèmes et leur provenance.

- **Gestion des erreurs IPC**

Les erreurs de communication inter-processus ont été traitées en ajoutant des mécanismes de **gestion d'exception** autour des opérations de lecture et d'écriture dans les queues de messages. Cela a permis de capturer des erreurs comme l'**EOFError** et de gérer la fermeture appropriée des queues, en s'assurant qu'aucun processus ne tente d'accéder à une queue fermée. De plus, la gestion des erreurs liées aux **connexions de socket** et aux messages a été améliorée en vérifiant régulièrement l'état de la connexion avant chaque opération.

- **Problèmes d'affichage**

Pour les problèmes liés au display.py, nous avons dû nous y reprendre à plusieurs fois, en repartant de la base, pour d'abord afficher les feux, récupérer l'alternance des feux, puis afficher les véhicules, les faire se déplacer, et enfin même prendre des virages (même si le rendu à ce niveau-là n'est pas propre). C'était, à ce niveau, surtout un problème d'organisation.

- **Problèmes de performance**

Pour les problèmes de **performance et de latence**, nous avons ajusté les **délais de synchronisation** pour mieux gérer les interruptions, en particulier lors du passage d'un véhicule prioritaire. Des **optimisations du temps de réponse** ont été faites en réduisant les cycles de mise en attente inutiles et en optimisant la gestion des queues de messages. Ces ajustements ont permis de réduire la latence et d'améliorer la réactivité du système en temps réel. Nous avons passé un peu de temps à comprendre pourquoi il fallait mettre des waits, mais ceux-ci nous ont beaucoup servi tout le long du code.

9. Tests et validation

a) Tests effectués

Pour valider la simulation, plusieurs **tests unitaires** ont été réalisés afin de vérifier le bon fonctionnement de chaque composant de manière isolée. En raison des premières difficultés rencontrées lors de l'intégration des différents modules, lors de laquelle nous avons tenté de tout programmer en parallèle, nous avons finalement adopté une **stratégie de tests individuels**. Chaque module a été testé séparément, avec des main temporaires créés avant d'être intégré dans le système global, ce qui a permis de mieux cerner les problèmes spécifiques à chaque partie du programme.

Les tests ont principalement porté sur :

- **La fluidité du trafic** : Vérification de la gestion correcte des véhicules qui circulent aux intersections, y compris le respect des règles de priorité (par exemple, priorité aux véhicules venant de la droite).
- **La gestion des priorités** : S'assurer que les véhicules prioritaires, comme les ambulances ou les véhicules d'urgence, étaient bien pris en compte par la simulation, avec les feux changeant immédiatement pour leur permettre de passer.
- **La gestion des feux de circulation** : Tester si les feux de signalisation changent de manière cohérente et respectent la logique demandée (par exemple, le passage au vert pour une direction avant de passer au rouge).

b) Résultats des tests

Au début, les tests ont révélé de **nombreux bugs**, principalement liés à la gestion des feux de circulation. La logique de changement des feux ne respectait pas toujours les priorités, et des conflits sur l'accès concurrent aux ressources partagées (comme les feux) ont été observés. De plus, la **simulation ne fonctionnait pas de manière fluide**, avec des délais et des latences plus importants que prévu. Après quelques ajustements de la synchronisation, la situation s'est améliorée, mais des **bugs subsistent**, notamment au niveau des feux, qui ne respectent pas toujours la logique de priorité ou changent à des moments incorrects.

Un autre problème soulevé par les tests concerne la gestion de la **queue de priorité** pour la gestion des véhicules prioritaires, qui semble ne pas fonctionner de manière optimale dans tous les cas. La gestion de l'ordre des véhicules n'est pas toujours fluide, ce qui entraîne des comportements imprévus dans la simulation. Nous avions aussi des problèmes de compatibilité des variables, qui faisaient que tout ne s'affichait pas (les sockets n'avaient pas le bon format).

c) Limites de la simulation

Certaines **fonctionnalités avancées** n'ont pas pu être implémentées pour des raisons de **complexité technique et de temps** :

- **Gestion des véhicules plus complexes** : L'ajout de comportements plus complexes pour les véhicules, comme les ajustements de vitesse, les différentes tailles de véhicules ou la gestion d'un plus grand nombre de véhicules, a été limité par la capacité de calcul et la gestion des processus.
- **Priorité à droite** : Bien que l'idée de la priorité à droite ait été envisagée, elle n'a pas été implémentée en raison de la complexité supplémentaire de synchronisation et de gestion des priorités de circulation.
- **Gestion des collisions** : La simulation ne gère pas les collisions entre les véhicules, ce qui aurait demandé des calculs géométriques supplémentaires pour détecter les interférences entre les trajectoires des véhicules.
- **Interface graphique** : Bien que l'interface ait permis une visualisation basique de la simulation, elle reste assez **rudimentaire** et manque d'esthétique et de fluidité.

Un des **bugs persistants** concerne la logique des **feux de circulation**. Les feux ne respectent pas toujours la logique de priorité et changent parfois de manière imprévue, en particulier dans des situations où des véhicules prioritaires devraient avoir la priorité, mais c'est assez aléatoire. Nous n'avons pas su identifier le problème. Peut-être que l'absence de lock sur la priority_queue a joué.

10. Conclusion

a) Bilan du projet

Ce projet a permis de mettre en place une simulation fonctionnelle de la gestion du trafic urbain avec des feux de circulation et des véhicules en interaction. Les **points positifs** incluent la séparation claire des différents modules, la gestion des processus parallèles et la coordination entre les modules. Cependant, nous sommes conscients qu'il reste beaucoup d'éléments à améliorer.

b) Améliorations possibles

Optimisation de la gestion des processus : L'optimisation de la gestion des verrous et des queues de messages pourrait améliorer la réactivité et la fluidité de la simulation.

Ajout de nouvelles fonctionnalités :

- Gérer un plus grand nombre de véhicules dans la simulation, en incluant des **comportements plus complexes** pour les véhicules, comme l'accélération, les trajectoires courbes ou les priorités dynamiques.
- Implémenter un système de **détection de collisions** pour rendre la simulation plus réaliste. Pour l'instant, une fois que la voiture dépasse son feu, elle n'est plus gérée (sauf pour la supprimer).
- Ajouter une gestion de **priorité à droite**, afin d'améliorer la fluidité de la circulation.
- Améliorer **l'interface graphique** en intégrant des animations fluides, des graphiques interactifs et des images.
- Tout lire via un seul main, un seul terminal, et non pas appeler display.py en parallèle.

c) Impact des choix techniques

Les choix techniques, comme l'utilisation de **multiprocessing** pour gérer les processus de manière parallèle et l'utilisation de **queues de messages** pour la communication inter-processus, ont eu un impact positif sur la **maintenabilité** du programme, en permettant de séparer les responsabilités et de faciliter les mises à jour et les ajouts de fonctionnalités. Toutefois, ces choix ont également introduit des **difficultés liées à la synchronisation**, qui ont rendu le développement plus complexe et ont nécessité des ajustements répétés pour garantir une exécution correcte des processus.

En conclusion, le projet a permis de démontrer les **principes fondamentaux** de gestion du trafic et de communication entre processus, tout en mettant en lumière des domaines d'amélioration pour optimiser la performance et étendre les fonctionnalités dans les futures versions du programme.