Ecole Publique d'Ingénieurs en 3 ans

Rapport

SIMULATION D'UNE PANDEMIE

Ayoub Goubraim `ayoub.goubraim@ecole.ensicaen.fr`,
Mohamed Ben lboukht `mohamed.ben-lboukht@ecole.ensicaen.fr`,
Omar Khali `omar.khali@ecole.ensicaen.fr`,
Hamza Ouazzani Chahdi `hamza.ouazzani-chahdi@ecole.ensicaen.fr`

Le 12 Janvier 2025

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

www.ensicaen.fr

Contents

1 Introduction

Le projet simule la propagation d'une contamination virale dans une ville, en intégrant divers acteurs comme les citoyens, pompiers et médecins. À travers cette simulation, nous utilisons des mécanismes de gestion des processus et de communication interprocessus pour modéliser et contrôler l'évolution de l'épidémie.

2 Gestion des citoyens

2.1 Gestion des Threads

Chaque citoyen et journaliste est simulé par un thread indépendant, créé via la fonction `create_citizen_threads`. Ces threads utilisent une structure de données dédiée (`citizen_thread_data`) qui contient des informations comme l'identifiant du citoyen, un pointeur vers la mémoire partagée (`shared_city`), et des outils de synchronisation tels qu'un mutex (`move_mutex`) et un sémaphore (`city_sem`). La fonction exécutée par chaque thread, `citizen_thread_function`, permet à chaque citoyen de calculer ses déplacements avec `calculate_new_position`, de vérifier leur validité avec `is_movement_valid`, et de mettre à jour sa position et son état en fonction des taux de contamination. Les threads fonctionnent de manière synchrone grâce aux sémaphores, qui contrôlent l'accès global à la mémoire partagée, et aux mutex, qui empêchent plusieurs threads de modifier simultanément les mêmes ressources.

2.2 Gestion de la Mémoire Partagée

La mémoire partagée est le centre névralgique du programme, permettant à tous les threads d'accéder aux données de la ville. Initialisée par la fonction `init_citizen_manager`, elle est configurée via `shm_open` et `mmap`, qui permettent de mapper un segment de mémoire dans l'espace du processus. La structure `city`, utilisée dans la mémoire partagée, contient la carte de la ville (`city_map`), une liste des citoyens avec leurs états et positions (`person_list`), ainsi que des statistiques globales (nombre de citoyens sains, malades, morts, etc.). Les threads accèdent à cette mémoire pour mettre à jour les positions des citoyens, lire les taux de contamination, et ajuster les statistiques globales à chaque tour. Ces modifications sont réalisées principalement dans les fonctions `update_citizen_state` et `process_turn`, qui s'assurent que les données partagées reflètent l'évolution de la simulation.

2.3 Gestion des Tours

La coordination des actions des citoyens et la communication avec le simulateur principal sont réalisées à chaque tour. Le programme reçoit des commandes via le pipe `pipe_from_epidemic` et envoie les statistiques mises à jour via `pipe_to_epidemic`. Les commandes incluent la mise à jour d'un tour (`MSG_UPDATE_TURN`), qui déclenche la fonction `process_turn` pour recalculer les taux de contamination et ajuster les statistiques globales, ainsi que l'arrêt de la simulation (`MSG_SIMULATION_END`), qui met fin à tous les threads en arrêtant la boucle principale. La fonction `process_turn` compile les statistiques globales et les envoie sous forme de structure `pipe_message`, assurant une communication claire et précise entre les processus.

2.4 Nettoyage des Ressources

Une fois la simulation terminée, la fonction `cleanup_citizen_manager` libère toutes les ressources utilisées par le programme. Elle attend la terminaison de chaque thread via `pthread_join`, libère la mémoire partagée avec `munmap`, ferme les sémaphores avec `sem_close`, et ferme les descripteurs de pipes ouverts. Ce processus garantit que toutes les ressources système sont correctement libérées, évitant ainsi les fuites de mémoire ou les conflits avec d'autres processus.

2.5 Conclusion

Le programme `Citizen Manager` offre une gestion multithreadée robuste pour simuler les interactions dans une ville. En combinant mémoire partagée, sémaphores et communication inter-processus, il fournit une simulation réaliste de la propagation épidémique.

3 Timer

Le programme Timer est un composant essentiel qui gère la progression des tours de simulation. Ses fonctionnalités sont décrites comme suit :

3.1 Initialisation

- Le Timer est initialisé avec deux paramètres :
  - **PID cible :** L'identifiant du processus du programme `epidemic_sim`.
  - **Durée de l'intervalle :** La durée (en secondes) de chaque tour, configurable entre 1 et 5 secondes. Si une valeur invalide est fournie, une durée par défaut de 1 seconde est utilisée.
- Les gestionnaires de signaux pour `SIGTERM` et `SIGINT` sont configurés pour permettre une terminaison propre du Timer.

3.2 Gestion des Signaux

- Le Timer envoie un signal `SIGUSR1` au processus cible à la fin de chaque intervalle, signalant la fin d'un tour de simulation.
- Les signaux de terminaison (`SIGTERM`, `SIGINT`) arrêtent le Timer et libèrent les ressources utilisées.

3.3 Fonctionnalité Principale

- Le Timer entre dans une boucle où il :
  - Envoie un signal `SIGUSR1` au processus `epidemic_sim`.
  - Attend la durée spécifiée à l'aide de la fonction `nanosleep`.
  - Gère les interruptions ou erreurs pendant l'attente et vérifie s'il doit continuer ou se terminer.

- Une gestion robuste des erreurs garantit un fonctionnement correct, même si le processus cible devient indisponible.

3.4 Terminaison

- Lorsqu'un signal de terminaison est reçu, le Timer :
  - Arrête d'envoyer des signaux.
  - Libère les ressources allouées pour éviter les fuites de mémoire.

4 Agence de presse

Le programme `press_agency` est conçu pour permettre la communication entre la simulation et l'interface via un tube nommé. Il reçoit et traite des messages tout en garantissant une gestion propre des ressources et des interruptions.

4.1 Fonctionnalités principales

- **Initialisation** : Le programme configure un système de journalisation des erreurs, ouvre un tube nommé en mode lecture et configure des gestionnaires pour les signaux `SIGTERM` et `SIGINT`.
- **Traitement des messages** : Les messages reçus peuvent être de différents types :
  - `MSG_STATS_UPDATE` : permettrait de traiter des mises à jour statistiques (non implémenté).
  - `MSG_SIMULATION_END` : arrête le programme de manière contrôlée.
  - Types inconnus : génère une erreur via la journalisation.
- **Gestion des signaux** : Les signaux `SIGTERM` et `SIGINT` permettent une interruption propre en mettant à jour un indicateur d'exécution (`running`).
- **Nettoyage** : Avant de quitter, les ressources allouées, comme le tube nommé, sont libérées pour éviter les fuites.

5 Epidemic Simulation

Le programme `epidemic_sim` initialise et lance l'exécution de la simulation d'épidémie en plusieurs étapes. Il commence par initialiser les composants nécessaires, tels que **le logger**, **les gestionnaires de signaux pour le timer** et **les processus de nettoyage**, et **a mémoire partagée** pour la ville. Ensuite, il crée **les pipes** nécessaires pour la communication entre les différents processus. Le programme initialise la ville et les citoyens, puis lance trois processus fils : **citizen_manager**, **press_agency**, et **timer**, qui gèrent respectivement la gestion des citoyens, la communication avec l'agence de presse et la gestion du timer pour les cycles de la simulation. Après cela, il crée un thread pour gérer le rendu graphique avec **SDL** et initialise **Gnuplot** pour afficher les statistiques de la simulation. Le programme entre ensuite dans une boucle où il attend les signaux pour avancer dans la simulation jusqu'à ce que le nombre de tours maximum soit atteint, avant de procéder au nettoyage et à la fermeture des ressources.

6 Interface Graphique

Cette section présente en détail les aspects techniques de l'implémentation de l'interface graphique, les solutions adoptées pour optimiser le rendu et les choix d'implémentation visant à assurer la fluidité et la modularité.

6.1 Organisation du rendu graphique

L'interface graphique utilise les bibliothèques SDL2 et SDL2_TTF pour une gestion optimale du pipeline de rendu. La structure principale `SimulationRenderer` centralise les ressources SDL essentielles, comme la fenêtre, le moteur de rendu, les polices et les données de simulation.

Le cycle de rendu suit une séquence stricte pour garantir des mises à jour fluides :

1. Effacer l'écran (`SDL_RenderClear`) avec une couleur de fond uniforme.
2. Parcourir les cases de la grille et appeler `render_cell`, qui applique une couleur spécifique selon le type de zone. Les citoyens sont affichés dynamiquement via `render_persons_in_cell`.
3. Afficher les informations textuelles (`render_info_panel`) et utiliser `render_text` pour le texte dynamique.
4. Dessiner la légende expliquant les couleurs (`render_legend`).
5. Présenter l'image mise à jour avec `SDL_RenderPresent`.

La gestion des ressources est rigoureuse pour éviter les fuites mémoire :

- Les textures et surfaces de texte sont détruites après utilisation.
- Les polices, le moteur de rendu et la fenêtre sont libérés proprement lors de la fermeture.

6.2 Optimisations et choix d'implémentation

Plusieurs optimisations et choix techniques ont été adoptés :

- **Abstraction des fonctions :** Des fonctions comme `render_cell` et `render_persons_in_cell` centralisent la logique d'affichage, réduisant la répétition de code.
- **Centralisation des constantes :** Les dimensions, couleurs et espacements sont définis comme constantes, facilitant les ajustements et évitant les incohérences.
- **Gestion des polices :** Une fonction générique, `render_text`, simplifie l'affichage du texte en gérant automatiquement les tailles et couleurs de police via SDL2_TTF.

Les principaux défis rencontrés incluent :

- **Alignement graphique :** Des ajustements répétés ont été nécessaires pour garantir une lisibilité optimale dans les panneaux d'informations et la légende.
- **Rendu dynamique :** L'affichage en temps réel des citoyens et des statistiques a été optimisé pour prévenir tout ralentissement.

- **Synchronisation des processus :** Parfois, l'interface ne parvenait pas à s'afficher correctement si certains processus de simulation n'étaient pas encore lancés. Pour résoudre ce problème, une fonction de synchronisation a été ajoutée. Cette fonction attend que tous les processus nécessaires soient opérationnels avant de lancer l'interface graphique.

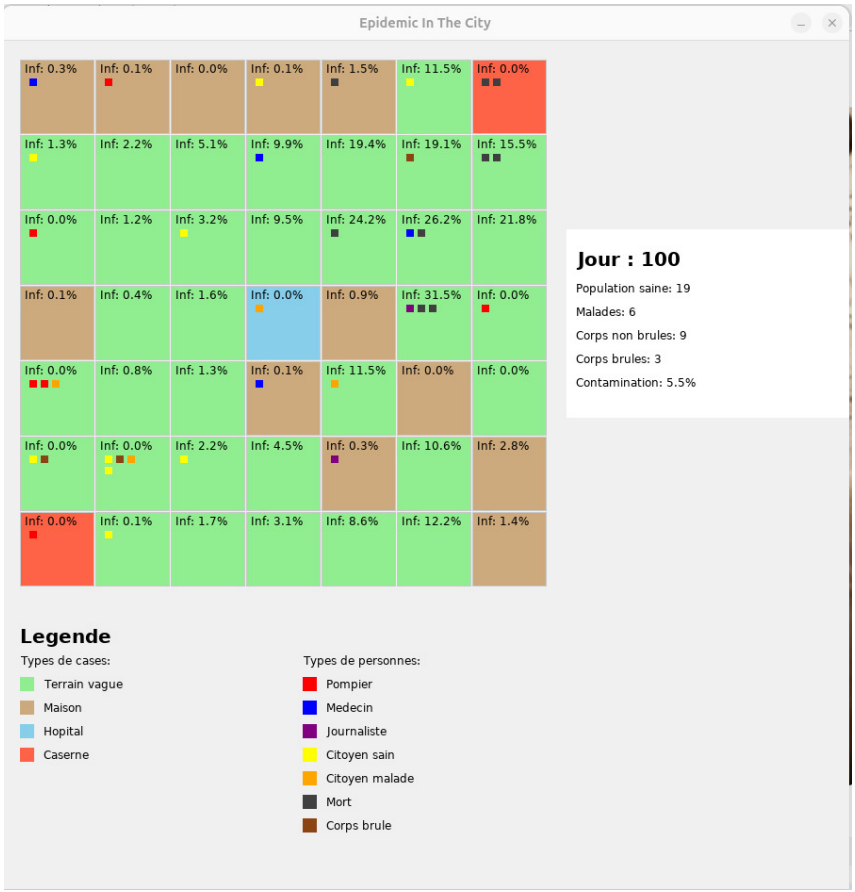

Figure 1: Notre interface graphique faite avec SDL

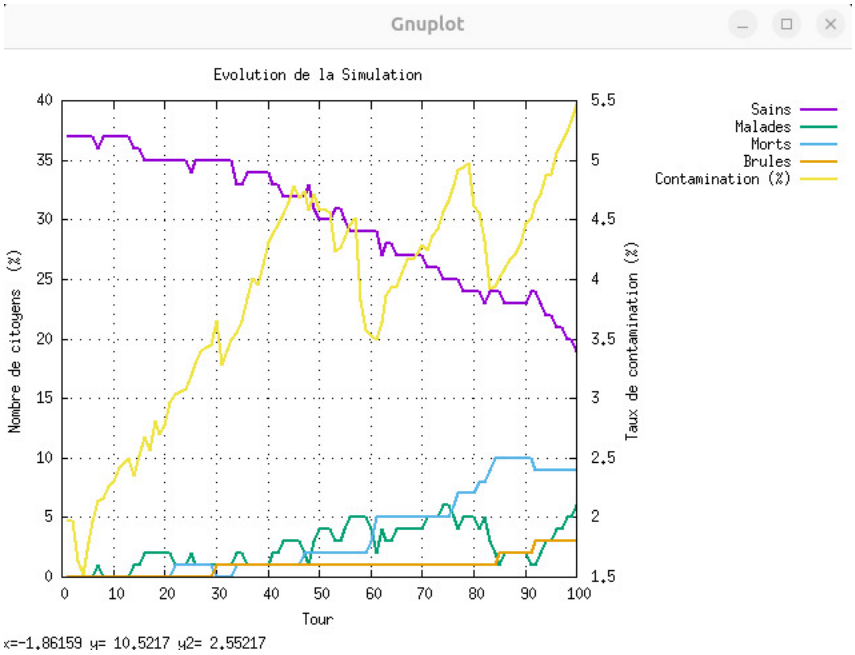

Figure 2:L'évolution des citoyens avec Gnuplot

7 Conclusion

Ce projet a été captivant à concevoir, bien qu'il ait nécessité un effort considérable et beaucoup de temps. Il nous a permis de mettre en pratique les concepts étudiés en cours de systèmes d'exploitation, tout en surmontant des défis techniques importants liés à la gestion des processus, de la mémoire partagée et des communications entre processus. Cette expérience a enrichi nos compétences et renforcé notre compréhension des principes de base des systèmes d'exploitation.

8 Ressources

Ce projet a été réalisé grâce à l'utilisation de plusieurs ressources en ligne et documentations. Voici les principales sources qui nous ont été utiles :

- **StackOverflow** : Pour la résolution de nombreux problèmes de programmation liés à l'utilisation de bibliothèques, la gestion des threads, des signaux et la communication inter-processus. Site web : `https://stackoverflow.com`
- **SDL Documentation** : Pour l'implémentation de l'interface graphique et le rendu avec SDL. Site web : `https://www.libsdl.org`
- **GNUPLOT Documentation** : Pour la génération des graphiques représentant l'évolution de la simulation. Site web : `http://www.gnuplot.info`
- **Linux Man Pages** : Pour les fonctions systèmes comme `shm_open`, `mmap`, et `sem_open`. Site web : `https://man7.org/linux/man-pages/`
- **Exemplier du Monsieur A.LEBRET** : Pour certains exemples de code relatifs à la gestion des processus en C. Site web : `https://github.com/alainlebret/os`