Ecole Publique d'Ingénieurs en 3 ans

Rapport

RAPPORT PROJET SYSTEME D'EXPLOITATION : LICENCE-TO-KILL

le 18 décembre 2022,
version 1.1

Ducastel Matéo,
Picque Kylian,
Seng Thomas,
Steimetz Tangui

Correcteur Projet : **Alain Lebret**

www.ensicaen.fr

TABLE DES MATIERES

1.1. Difficultés Rencontrées et Solutions. _____ 3

1.2. Différentes aides extérieures utilisées. _____ 3

2. Choix de conceptions des différents programmes. _____ 4

2.1. Mémoire partagée._____ 4

2.2. Spy_simulation._____ 4

2.3. Citizen_manager. _____ 4

2.4. Enemy_spy_network. _____ 5

2.5. Counter_intelligence_officer. _____ 5

2.6. Enemy_country. _____ 6

2.7. Timer. _____ 6

2.8. Autre. _____ 7

INTRODUCTION

1.1. Difficultés Rencontrées et Solutions.

De nombreuses variables globales ont été utilisées pour simplifier l'implémentation des signaux et leurs traitements respectifs, notamment l'utilisation d'un booléen *runnable* pour stopper les différents processus et threads et en modifiant sa valeur lors de la réception du signal *-SIGTERM*.

De plus, les programmes principaux possèdent le pointeur sur la mémoire partagée en global car la lecture des données et la fermeture de la mémoire par les threads ne s'effectuaient pas correctement, ce qui provoquait des *seg-fault* et des fuites. De plus, d'autres arguments sont en global, tels que les informations liées aux threads pour qu'ils aient tous accès aux mutex par exemple. Nous n'avons pas trouvé de solution plus efficace.

Les phases de Tests des différents programmes étaient compliquées car ces derniers forment un corps et les tester indépendamment des autres était une tâche difficile. Ainsi, nous avons utilisé différentes solutions :

- La bibliothèque *criterion* pour effectuer des tests unitaires sur les fonctions qui sont testables ;
- Des scripts bash pour simuler les signaux en continu et/ou un et un terminal supplémentaire pour générer les signaux « rares ».

De plus, l'utilisation massive de signaux entraine certaines confusions et problèmes d'implémentation difficiles à résoudre.

D'autre part, nous avons tenté d'utiliser les fonctions pour envoyer des signaux à un thread, en particulier les signaux qui servent aux combats. La solution trouvée a été laisser la gestion des signaux au processus père.

D'après *valgrind*, le programme *monitor.c* possède des fuites de mémoire liées à la bibliothèque *ncurses*. Celles-ci n'ont pas pu être résolues.

1.2. Différentes aides extérieures utilisées.

Utilisation de différents sites tels que *Stack Overflow* ou *Koor* pour savoir comment utiliser certaines primitives et fonctions.

Utilisation d'une nouvelle IA, *chatGTP*, de *OpenAI* qui nous a permis de voir comment trouver et utiliser certaines solutions. Celui-ci a été utilisé comme un outil d'aide et non de création.

2. Choix de conceptions des différents programmes.

2.1. Mémoire partagée.

Le projet a commencé par l'initialisation de la mémoire partagée. Nous avons fait le choix de créer un fichier qui s'occupe de son initialisation, sa création, sa destruction, son chargement et son déchargement, à l'aide de fonctions basiques pour simplifier les multiples utilisations dans les différents programmes. L'appel de la fonction de création de la mémoire partagée effectue toutes les étapes pour créer la carte et assigner correctement les personnages à un foyer et une entreprise.

L'accès à la mémoire partagée a été permis via des sémaphores afin qu'il soit contrôlé. Par exemple, lors du déplacement d'un personnage, il faut modifier dans la mémoire partagées le nombre de personnes qui se trouvent sur les cases de départ et d'arrivée.

2.2. Spy_simulation.

Ce programme est utilisé pour lancer les différents processus, qui vont se mettre en attente, pour initialiser la simulation. Il permet de mettre en relation le Timer qui synchronise la simulation et les processus qui attendent d'être notifiés. De plus, il renseignera la fin de la simulation et déclenchera la destruction des processus. La synchronisation et la communication sont faites via les signaux *-SIGALARM, -SIGCONT et -SIGTERM*, ce qui permet d'isoler les différents programmes et de les rendre indépendants lors de l'implémentation.

2.3. Citizen_manager.

La gestion des citoyens est lancée par la simulation à l'aide de la primitive *exec*. Le choix a été fait d'utiliser 127 threads, où chaque thread est considéré comme un citoyen unique. Cette modélisation permet de coder la vision du citoyen comme une entité indépendante. Les arguments donnés aux threads représentent l'état à un instant précis de la simulation, afin de rester cohérent avec l'implémentation décrite précédemment. Des arguments globaux sont utilisés afin de synchroniser les threads et mettre en exergue la partie commune de ceux-ci. Le default de cette modélisation est la vérification du bon fonctionnement de 127 threads, problème résolu par débogage de chaque thread.

Chaque thread est appelé à exécuter son traitement de tour lorsque le programme principal reçoit un signal, *-SIGCONT*, et réveille tous les threads, *pthread_cond_broadcast*, qui se remettront en veille après le traitement, ce qui permet une bonne synchronisation des citoyens tout en restant indépendant. Chaque thread bloque, à l'aide d'un sémaphore, dès qu'il traite

l'action, car nous avons rencontré des problèmes en essayant de faire des sémaphores précis / ciblés lors de la lecture / écriture dans la mémoire partagée.

2.4. Enemy_spy_network.

Officier traitant du réseau d'espionnage :

Afin de gérer les actions de l'officier traitant du réseau d'espionnage lors de la journée, le choix a été fait de séparer la gestion de celles-ci. Dans un premier temps, nous allons mettre à jour les différents temps de d

Lors de sa création, la file de message est tout d'abord réinitialisée afin de supprimer les potentiels messages qui s'y trouveraient, dû à une précédente simulation, puis lors de l'envoi des messages, elle est ouverte et nous pouvons y envoyer les messages stockés dans les données des threads.

Le réseau d'espions :

Pour le réseau d'espionnage, nous avons fait le choix de travailler avec des états. En effet, les espions ont une succession de fonctions selon l'état dans lequel ils sont actuellement. On a implémenté des états pour le jour et pour la nuit pour faciliter l'utilisation du *timer*.

Les threads qui dirigent les espions n'ont pas le contrôle des signaux. Cela permet d'éviter des répétitions de code mais aussi de simplifier l'utilisation des signaux. L'inconvenant de cette méthode est que nous somme obliger d'utiliser la mémoire partager pour préciser quel espion a été touché.

2.5. Counter_intelligence_officer.

L'officier de contre-espionnage :

Le programme de l'officier du contre-espionnage n'utilise pas de thread, un processus unique s'exécute.

Afin de connaitre l'avancement dans la partie de l'officier, la gestion des actions de l'officier du contre-espionnage se fait par la mise à jour de booléens représentant des statuts. Au début du tour, s'il s'agit d'une nouvelle journée, il détermine l'heure à laquelle il part vérifier la boite aux lettres dans l'éventualité où il connaitrait sa position. Puis, une vérification des points de vie de l'officier est effectuée, afin de savoir s'il doit fuir et finir la partie.

Certaines informations utiles sont également stockées en dehors de la mémoire partagée, comme les informations des caméras, le statut de l'officier concernant son avancement dans la partie, ou encore le nombre de jours qu'il a consacré à l'inspection d'un bâtiment pour trouver la boite aux lettres.

Les Caméras :

Celles-ci sont créés et détruites depuis l'officier de contre-espionnage, auteur du lancement de la vérification de la présence de personnages suspects sur la carte lorsqu'il se trouve à la Mairie, à l'aide d'un signal *-SIGUSR1*. Cela permet d'éviter que les caméras ne soient trop gourmandes pour le CPU et dans l'éventualité qu'elles se désynchronisent. Celles-ci vérifient qu'il n'y ait personne dans la rue à des horaires suspectes (1h à 5h) et si un personnage est repéré deux fois aux alentours d'une entreprise aux heures « piles » consécutives. C'est une façon simple de les implémenter, sans qu'elles ne soient trop performantes dans la détection d'espion mais qu'elles puissent également suspecter des citoyens.

2.6. Enemy_country.

Le pays ennemi a simplement pour rôle de réceptionner les messages de la file de message. Pour se faire, il va ouvrir la même file de message que l'officier traitant en lecture seulement et réceptionner les messages qui lui parvienne. Lorsqu'un message est reçu, il vérifie que celui-ci est correct (donc que ce n'est pas un faux message) et l'écrit dans un fichier qui sert de stockage des messages reçus.

Nous pouvons vérifier que de nouveaux messages sont présents dans la file via la structure mq_attr, qui contient un champ mq_curmsgs nous donnant le nombre de messages dans la file.

2.7. Timer.

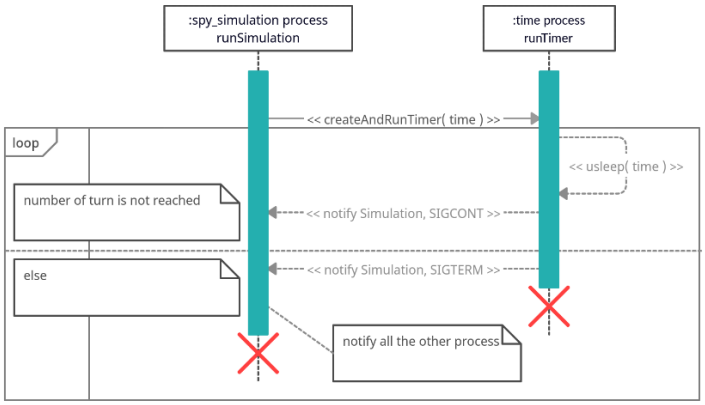

Figure 1 - Diagramme de Séquence du fonctionnement global du Timer.

Le Timer est créé par la simulation et sert à synchroniser la simulation en notifiant le gestionnaire de simulation à l'aide de signaux. Le Timer envoie *-SIGALARM* lorsque qu'un tour débute, et *-SIGTERM* pour signaler la fin et la destruction du Timer. L'utilisation des signaux permet de séparer le plus possible leurs interactions, sans surutiliser la mémoire partagée dans le but de faire un Timer le plus général possible.

L'appel à *usleep* a été utilisé pour créer notre propre alarme car nous avons rencontré des problèmes avec l'utilisation de la primitive *-SIGALARM*, notamment à cause du linker à la compilation et la transmission des temps aux programmes supérieur. De plus, par soucis

d'implémentation et de temps, c'est le Timer qui met à jour les temps directement dans la mémoire partagée ce qui rend la généralisation du Timer compromis.

2.8. Autre.

Récupération d'un personnage :

Le but est de pouvoir récupérer un personnage ou sa localisation sous forme de *point_t* / *cell_t* simplement à l'aide de son l'ID. Ces fonctions partent du principe qu'il faut éviter que la fonction appelante ait connaissance du type de personnage. Ces fonctions retournent un *(void*)* ou un pointeur sur une position, car le polymorphisme n'est pas faisable en C. Ces fonctions sont notamment utilisées par les caméras et l'officier de contre-espionnage pour repérer ou suivre un personnage.

Path-finding :

Un premier algorithme, optimisé, qui effectue un parcours en largeur afin d'obtenir le plus court chemin. Celui-ci est utilisée quand le personnage requiert d'être rapide, comme pour l'officier de contre-espionnage.

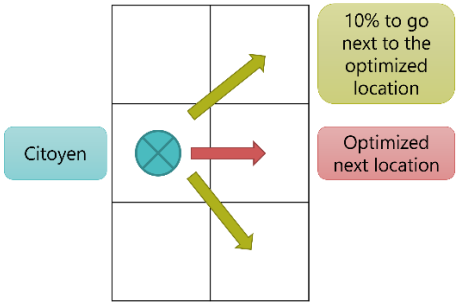

Figure 2 - Visualisation du déplacement aléatoire des citoyens.

Le second algorithme pour les déplacements qui ne requièrent pas d'être optimisé. Afin de rendre la simulation plus aléatoire, nous avons également implémenté le fait que le personnage ne choisit pas tout le temps le chemin le plus optimisé. 20% du temps, celui-ci va se rendre sur une case adjacente à la case optimisée (Cf. figure 2).

Lors de l'implémentation du chemin non déterministe nous avons rencontré quelques problèmes notamment sur la gestion des erreurs lorsqu'une case n'est pas accessible. Ceci a été corrigé à l'aide de test unitaires et de *criterion*, qui ont permis de vérifier tous les différents cas possibles.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES