Rapport

A License To Kill

le 7 janvier 2024,
version 1.1

Cécile LU,
Paul NGUYEN,
Antonio CIMINO

Professeur : Alain LEBRET

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

www.ensicaen.fr

Table des matières

0. Introduction

Ce projet consiste en une simulation de ville dans laquelle cohabitent des citoyens, des espions et leur officier qui travaillent pour un pays ennemi et un officier de contre-espionnage qui réside et travaille à la mairie. Chaque habitant a un appartement dans une résidence, un travail et peut se rendre au supermarché, L'objectif des espions est de voler des informations aux entreprises de la ville et de les transmettre au pays ennemi.

Dans ce rapport, nous présentons les différents programmes qui communiquent entre eux dans le cadre de la simulation, puis les difficultés que nous avons rencontré.

I. Implémentation de la simulation

1.1. Spy Simulation

Ce programme est le point central de la simulation, il va d'abord créer la mémoire partagée, initialiser toutes les informations sur les entreprises, la carte et les coordonnées des emplacements importants, l'heure etc. Il va ensuite se cloner avec `fork()` et recouvrir avec `execl()` plusieurs fois, une fois pour chaque autre programme dont la simulation a besoin. On stocke notamment dans la mémoire partagée, le 'pid' de chaque processus associé à un programme pour faciliter l'envoie de signaux entre les processus. A chaque tour, le programme reçoit un signal de 'timer' et appelle une fonction `next_turn()`. Cette fonction met à jour la mémoire partagée selon notamment l'heure, l'état des citoyens etc. puis signale tous les autres programmes à la fin du tour.

1.2. Personnages de la simulation

1.2.1. Citizen Manager

Le module 'citizen manager' de notre projet simule la vie quotidienne des citoyens dans un environnement urbain. Chaque citoyen est créé dans la mémoire partagée et est associé à un thread distinct. On initialise les attributs d'un citoyen au début de la simulation, un lieu de résidence et un lieu de travail sont notamment assignés de manière aléatoire. Cette conception permet de simuler leurs activités quotidiennes telles que le travail, les achats et le retour au foyer. Quand le processus reçoit un signal pour dire que la simulation est passée au prochain tour, tous les threads sont mis à jour à l'aide de la fonction `pthread_kill()`. Cette mise à jour permet au citoyen, en fonction de l'heure de la journée, d'effectuer le bon déplacement et d'être dans l'état attendu. La création des entreprises (spy_simulation) et l'assignation des citoyens à une entreprise (citizen_manager) est réalisée manuellement au début de la simulation. Nous avons fait ce choix pour assurer une diversité de tailles d'entreprises.

1.2.2. Enemy Spy Network

Dans le module ‘enemy spy network’, la simulation est centrée sur les activités des espions. Ils sont classés en deux catégories : les espions classiques et l'officier d'espionnage. Chaque personnage est initialisé en début de simulation avec des attributs définis dans la mémoire partagée. Les espions sont affectés à des threads exécutant des routines et les signaux sont gérés de la même manière que pour ‘citizen manager’. En fonction de l'heure dans la simulation, l'espion pourra soit aller en repérage, voler, faire ces courses, ou rester chez lui. Leur mission est de cibler des entreprises pour y dérober des informations et déposer des messages dans une boîte aux lettres spécifique. Les messages récupérés sont déjà chiffrés (chiffrement de César) et contenus dans les entreprises. Le processus de vol d'informations s'étend sur 18 tours, comprenant une phase de repérage autour de l'entreprise ciblée et une phase de vol proprement dite. Pour cibler une entreprise, nous avons décidé que parmi toutes les entreprises, il allait en choisir une aléatoirement. Le vol dure 6 tours avec 5 tours où il reste dans l'entreprise, et au 6eme tours, on choisit un message de l'entreprise de manière aléatoire qui sera récupéré par l'espion (Le choix aléatoire et la distribution des messages d’importance différentes dans les entreprises simule le comportement attendu en terme de probabilité d’importance du message volé). L'officier d'espionnage se concentre sur la collecte et la transmission de messages via la file de messages. La transmission se fait avec les messages toujours chiffrés, tous les jours à 23h depuis le domicile de l’officier. Les espions interagissent avec un officier de contre-espionnage qui affecte leur santé via des signaux spécifiques qui est le Counter Intelligence Officer.

1.2.3. Counter Intelligence

Contrairement aux autres personnages, l'officier de contre-espionnage n'utilise pas de threads. Pour accéder à la mémoire partagée, nous devons utiliser des sémaphores. Son rôle dans notre programme est simple : détecter toute activité suspecte autour d'une entreprise. Les activités suspectes sont représentées par la présence d’un personnage dans une entreprise la nuit. Nous supposons que les entreprises ont des caméras qui détectent la présence de quelqu’un à une heure anormale (Cette caméra est activée par le thread espion quand il pénètre dans l’entreprise la nuit). S'il repère quelque chose de suspect, l’officier de contre-espionnage se déplace rapidement vers cet endroit, suit l'espion et cherche la boîte aux lettres. Pour suivre l’espion, il accède à la mémoire

partagée et récupère l'identifiant de l'espion en question, et ainsi accède à ses coordonnées à chaque tour. Une fois la boîte aux lettres trouvée, sa routine quotidienne est simple : se rendre à la boîte aux lettres à une heure aléatoire de la journée. En cas de rencontre avec un espion, il doit soit se battre, soit s'enfuir. Lorsqu'il se bat, il envoie un signal SIGUSR2 à l'espion avec lequel il se confronte.

1.3. Timer

Ce programme utilise la fonction `us_sleep()` provenant de l'exemplier, pour temporiser le temps d'un tour de simulation. Ce programme a également un argument qui désigne la durée d'un tour en millisecondes. On utilise des signaux pour communiquer avec `spy_simulation`, et indiquer la fin d'un tour ou la fin de la simulation (si on a atteint 2016 tours, ou si la mémoire partagée indique la fin de la simulation). Les signaux utilisés sont les signaux `SIGTTIN` pour la fin d'un tour et `SIGTERM` pour la fin de la simulation.

1.4. Monitor

Le programme 'monitor' nous a été partiellement donné et nous avons complété l'affichage des différentes informations de la simulation dans les fonctions du fichier 'monitor.c'.

1.5. Enemy Country

Ce programme met en place une file de messages pour permettre l'envoie de messages avec une priorité entre `enemy_spy_network` et `enemy_country`. Une fois la file de messages mise en place, on attend de recevoir les messages pour en placer le contenu déchiffré dans la mémoire partagée et qui sera également affiché par le programme 'monitor'.

2. Difficultés rencontrées

2.1. Difficultés à coder

Notre première difficulté résidait dans l'amorce de notre processus de codage. Bien que nous disposions d'une base de code, trouver une vision d'ensemble pour notre programmation a pris beaucoup de temps. Nous avions envisagé initialement de commencer par coder le citizen manager, pensant que cela serait plus simple. Cependant, l'absence de base pour ce programme nous a contraints à imaginer entièrement la structure du code, compliquant ainsi notre tâche initiale. Nous avons donc décidé de repartir de zéro pour rendre le programme plus accessible. Ayant déjà traité les threads dans ce processus et défini une forme générale pour un programme, coder par la suite les espions et le contre-espion a été plus aisé sur le plan structurel.

Notre deuxième difficulté majeure a été de déterminer la gestion des threads. Nous avons opté pour la création des personnages en mémoire partagée et leur association à des threads à l'aide d'un tableau de structure contenant l'ID du thread et l'ID du citoyen ou de l'espion correspondant, ce qui permet à chaque thread de mettre à jour le personnage qui lui est associé (notamment grâce à `pthread_self()`). De plus, pour faciliter la synchronisation des threads, dans 'citizen_manager' et 'enemy_spy_network', nous avons mis en place une barrière de thread, qui permet à chaque thread de trouver le citoyen ou l'espion qui lui est associé, puis quand tous les threads sont prêts, de les mettre à jour en même temps.

Certains problèmes nous ont pris beaucoup de temps. Notamment l'affichage des messages par monitor dans le 'enemy country monitor', qui peut encore parfois causer des problèmes de mémoire partagée et faire crasher la simulation.

La fin de la simulation peut potentiellement causer des fuites de mémoire, nous avons cependant fait en sorte de terminer correctement tous les threads et de libérer la mémoire occupée par la mémoire partagée avec `shm_unlink()`, et par la file de messages.

2.2. Éléments manquant

Nous n’avons pas pu mettre en place certaines mécaniques de la simulation pour des raisons techniques ou organisationnelles. Par exemple, pour que la simulation soit plus complète, il nous faudrait gérer plus précisément les interactions entre les espions et l’officier de contre-espionnage avec les signaux pour simuler les dégâts infligés. Il nous manque également la gestion des événements de fin de simulation lorsque l’espion s'enfuit après une altercation avec l’officier de contre-espionnage.

3. Conclusion

Ce projet a été très enrichissant en termes de connaissance technique sur la communication interprocessus. En explorant les files de messages, les sémaphores et les zones de mémoire partagée, nous avons saisi l'importance cruciale de ces outils pour des échanges efficaces entre processus. Ce projet nous a permis de relever les défis en lien avec la synchronisation, l'accès aux ressources partagées et la gestion de la mémoire dans le contexte de la communication entre processus.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE