Programmation Concurrente

César COLLÉ

Rémy KALOUSTIAN

POLYTECH NICE-SOPHIA

SI4

10/10/2016

1. Introduction

2. Algorithme de déplacement

3. Java vs Posix

4. Algorithme de création des processus fils

5. Algorithme de terminaison de l'application

6. Conclusion

1. Introduction

L'objectif de ce projet est de simuler le déplacement de personnes sur une grille et ce, à l'aide de threads qui géreront le déplacement des personnes. Outre l'objectif fonctionnel, nous avons un objectif pédagogique qui est de bien nous rendre compte de l'utilité et du fonctionnement des threads.

Nous avons choisi d'implémenter ce projet en C car le choix d'un langage orienté objet ne nous semblait pas nécessaire (pas de polymorphisme ni d'héritage).

2. Algorithme de déplacement

Précondition : toutes les entités de la carte sont situées dans une zone libre.

Post-condition : L'entité se déplacera obligatoire vers la gauche. Ou elle se met en attente.

Variables : P – Décrit une personne sur la carte

Algorithm 1 : Algorithme de déplacement d'une entité

```
if(dans_limites_du_thread() && Statut(P) == disponible)
    if(person.y < limite_milieu1)
         if(case_bas_gauche_libre())
               bouger_bas_gauche() ;
         else if(case_gauche_libre())
               bouger_gauche() ;
         else if(case_bas _libre())
               bouger_bas() ;
    else if (person.y> limite_milieu2)
         if(case_haut_gauche_libre ())
               bouger_haut_gauche() ;
         else if(case_gauche_libre())
               bouger_gauche() ;
         else if(case_haut_libre())
               bouger_haut() ;
    else if (person.y > limite_milieu1 && person.y < limite_milieu2)
         if(case_gauche_libre())
               bouger_gauche() ;
```

3. Java vs Posix

Création : En C, on appelle pthread_create() en lui passant en paramètre le thread, la fonction à exécuter, et des informations complémentaires sur le comportement du thread. Une majeure partie de la création se fait via des paramètres et on a juste à déclarer une variable de type pthread_t.

En Java, où l'on doit tout d'abord créer une classe héritant de Thread, qui représentera notre thread. La fonction à exécuter est déjà dans la classe (run()). Contrairement au C, il n'y a pas de paramètres à passer lors de la création, qui se fait comme une instanciation d'objet classique. C'est une solution rapide dans certains cas.

Le gros problème avec cette méthode c'est que le langage java n'autorisant pas l'héritage multiple, on perd donc la chance d'utiliser pleinement la puissance de Java et du polymorphisme.

Il y a une deuxième façon de procéder à la création d'un thread en java. En effet on peut aussi effectuer l'implémentation d'une interface Runnable que l'on passera au constructeur de la classe Thread. Cette méthode est la plus générale car elle permet en outre de pouvoir hériter d'une autre classe et permet de n'avoir qu'à écrire le code de la fonction run().

Nous pensons qu'utiliser l'interface Runnable est donc un choix plus judicieux, pour sa capacité de réutilisation.

Démarrage : L'appel à pthread_create() crée et démarre le thread. Cependant, si nous ne faisons pas rejoindre le thread créé et le programme principal, il y a de fortes chances que ce dernier se termine sans que le thread créé n'ait pu s'exécuter en entier. Pour éviter cela et assurer le bon déroulement du programme, nous utilisons pthread_join(), en passant le thread en paramètre.

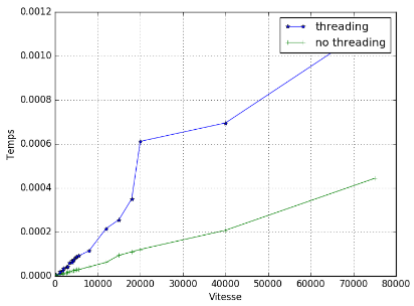
En Java, il suffit dans les deux cas d'appeler la méthode start() de l'objet Thread, qui produit un appel à la méthode run() par la JVM. On peut aussi ajouter la méthode join() permettant au thread appelant d'attendre la fin de ses fils.

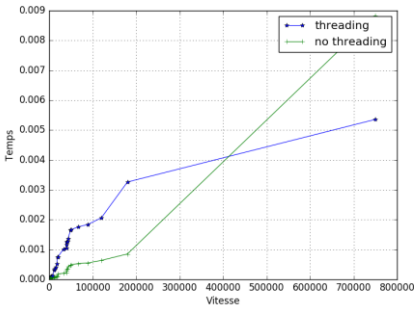
Arrêt : En C, on peut appeler pthread_exit() qui va arrêter l'exécution du thread et revenir au thread principal. Cette fonction est par ailleurs appelée implicitement à chaque fois qu'un thread à terminer son exécution.

En Java, lorsque le thread termine son traitement il se finit. On peut aussi le stopper avec l'appel de la méthode stop().

Destruction : En C, on peut utiliser pthread_cancel() qui envoie une requête de destruction au thread. Il se peut que cela ne détruise pas le thread, la destruction dépendra de deux facteurs, l'état et le type de destruction qui peuvent être choisis avant la destruction.

Java étant un langage à pile. Le garbage collector de la JVM détruit le thread ainsi que tous les composants qu'il aurait alors créés





4. Algorithme de création des processus fils

Pre-condition : N > 1

Invariant : k valeurs de 0 à N

Variables : pthread[] threads, tableau de threads

boundary[] boundaries, tableau contenant la liste des limites pour les threads

version : version dénote la version de l'application (0, 1, 2) et prend les valeurs associés i.e le nombre de thread necessaire à chaque version.

precondition :

Chaque processus contient des limites de traitements que ça soit un quart de la carte pour le t1 ou toute la carte pour le t0. Nous n'avons pas de limutes de traitements pour le t2

```
create_boundaries() ;
for k allant de 0 a version do
      create_thread(boundaries[k]);
      join(threads[k]);
end
```

5. Algorithme de terminaison de l'application

Comment on termine notre application

Variable : P → liste des personnes disponibles.

Pré-condition : Il y a une personne qui est disponible

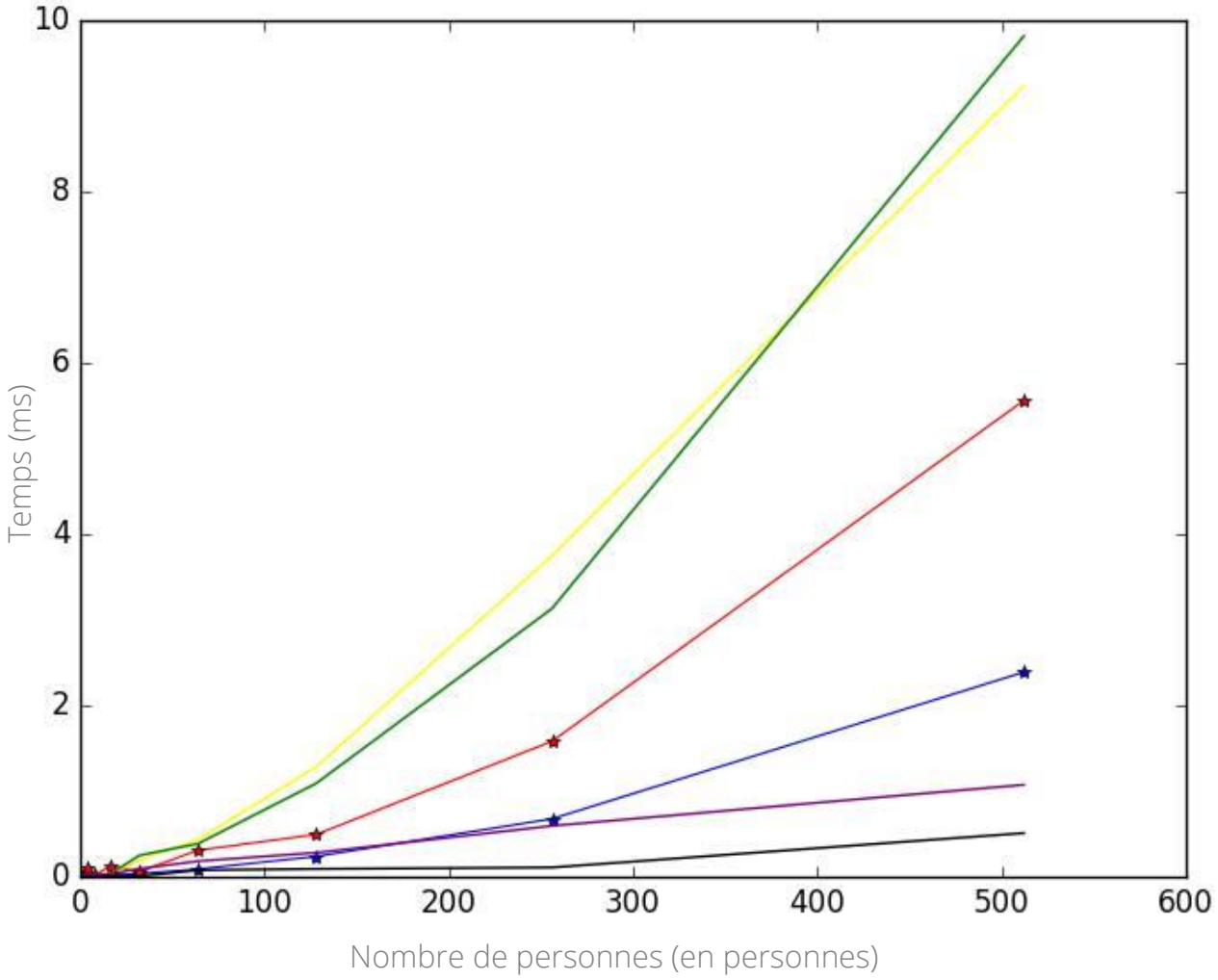
Variables : person * population, liste des personnes sur la grille

int nbpeople , nombre de personnes sur la grille

Invariant : En vertu de la post condition du déplacement qui est qu'on doit automatiquement se déplacer à gauche et que la position finale où s'arrête le processus est à gauche, alors il existe un chemin C tq pour tout processus p , p prend C et peut finir.

```
tant qu'il y a des personnes disponibles
        execute_thread_movement(P) ;
end
terminate_program() ;
```

6. Comparaison des scénarios



Légende:

- Temps utilisateur pour –t0
- Temps CPU pour –t0
- Temps utilisateur pour –t1
- Temps CPU pour –t1
- Temps utilisateur pour –t2
- Temps CPU pour –t2

On constate dans tous les cas que le temps utilisateur est inférieur au temps CPU. De plus le scénario le moins chronophage est celui ayant un thread par personne.

7. Conclusion

Au niveau des choses à améliorer, nous pourrions peut-être optimiser notre code afin d'effectuer des opérations en moins et ainsi consommer moins de temps. D'après nos mesures, nous avons remarqué que le mode -t1 consomme le plus de temps utilisateur et de temps cpu. Le choix de la meilleure option est partagé entre –t0 et –t2, le premier étant plus rapide au niveau du temps utilisateur et le deuxième lui, plus rapide an niveau du temps cpu. Nous voyons ainsi que pour le cas d'une grille séparée en quatre, l'exécution est plutôt lente, là où elle est rapide avec un seul thread ou un thread par personne.

Le rapport doit être rédigé comme un rapport et donc comporter outre les éléments attendus une introduction et une conclusion. Dans cette première étape, il doit insister et décrire : - l'algorithme utilisé pour déplacer une personne (rappel : un algorithme n'est pas le code C). Pour ceux qui ne savent pas ce qu'est un algorithme vous pouvez lire l'article : https://interstices.info/jcms/c_5776/qu-est-ce-qu-un-algorithme.

- comparer la manipulation des threads en Java (que vous avez vu en cours en SI3) et la manipulation des threads Posix (création, démarrage, arrêt, destruction, passages de paramètres, terminaison) ;

- pour la thread principale (i.e. celle associée au main de l'application), vous devez donner l'algorithme de création des threads filles (option –t1 et –t2) et celui lié à la terminaison de l'application (i.e. attente de création des threads filles précédemment créées) ;

- analyser la correction de chacun des scénarios proposés.

- analyser de manière comparative les divers scénarios corrects proposés, cette analyse doit nécessairement utiliser les mesures effectuées.

+ Dis en cours (titre, date, auteurs, introduction, ce qu'on a fait (gné) Bilan (bien, moins bien, lent, rapide, on a détecté...))