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

2. Algorithme de déplacement

Explication de notre algo de déplacement.

Nous avons implémenté pour le moment un algorithme simple de déplacement qui créé un déplacement d'une entité en premier temps vers le premier orifice de la carte du jeu puis vers un second orifice.

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, 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

RAJOUTER UN EXEMPLE

Le gros problème avec cette méthode c'est que le langage java n'autorisant pas le multiple héritage, 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 général car elle permet en outre de pouvoir hériter d'une autre classe et permet de réutiliser le code de l'interface.

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.

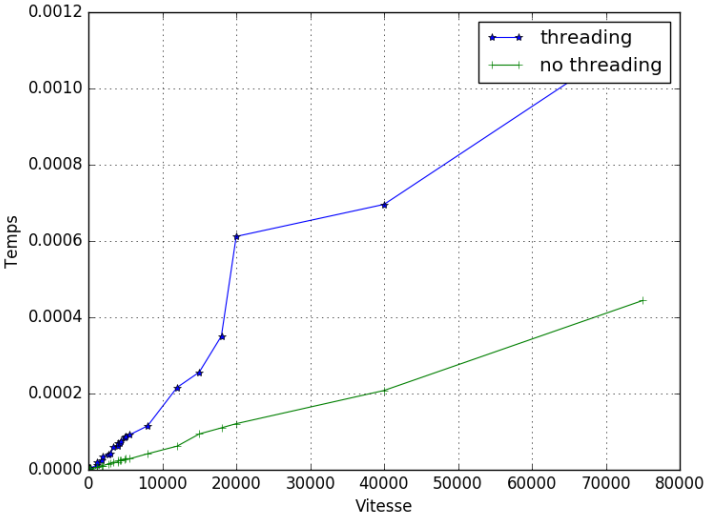
En Java, il suffit dans les deux cas d'appeler la méthode start() de l'objet Thread, qui produit un appel à la méthode run() par la JVM. On peut aussi ajouter la méthode join() permettant au thread appelant d'attendre la fin de ses fils.

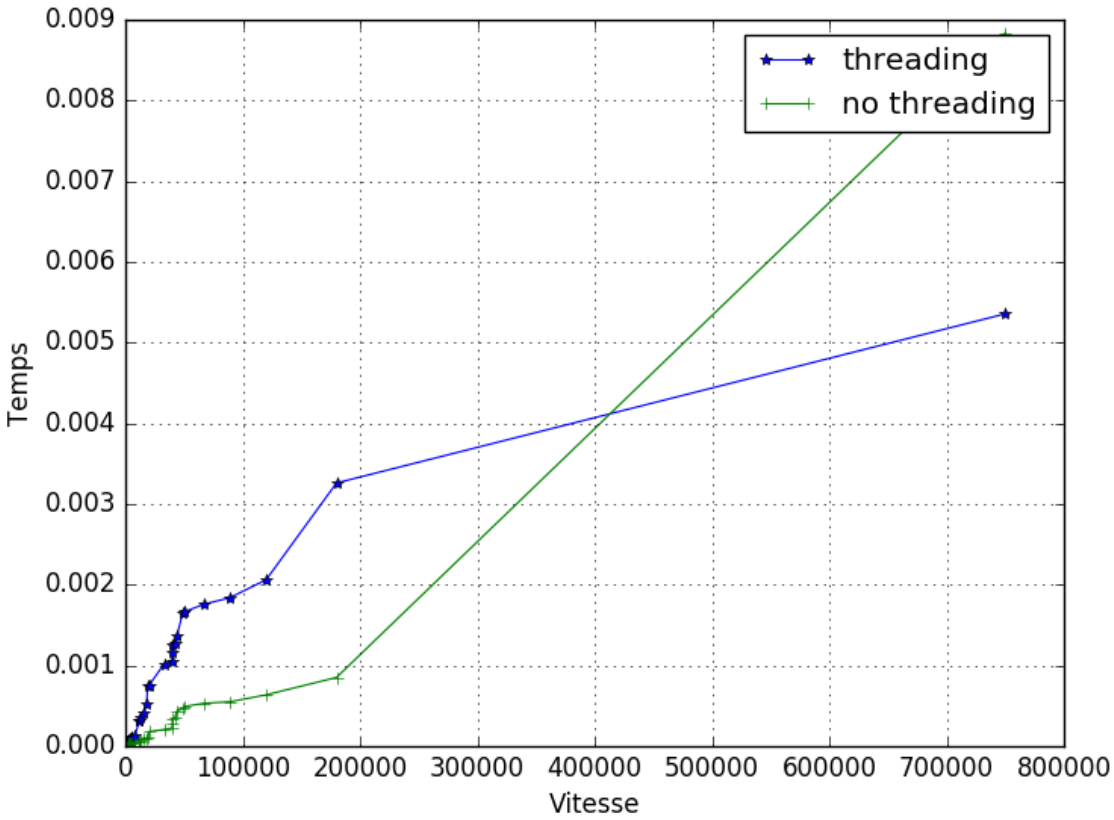
Arrêt : On peut utiliser pthread_cancel() qui envoie une requête de destruction au thread. Il se peut que cela ne détruise pas le thread, la destruction dépendra de deux facteurs, l'état et le type de destruction qui peuvent être choisis avant la destruction.

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

Destruction : //MANQUE POUR C

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





4. Algorithme de création des processus fils

Comment on crée nos fils.

Cas single thread :

Pre-condition : N > 1

Invariant : k valeurs de 0 à N

1) Partage de la carte en N sections numéroté de 0 à N-1.
2) Pour tout k allant de 0 à N : créé un processus k
3) Attribution au processus k sa portion numéro k de la carte.

Version adaptée

Variable : pthread[] threads, tableau de threads

```
create_boundaries() ;
for k allant de 0 a boundaries.size() do
      create_thread(boundaries[k]);
      join(thread);
end
```

5. Algorithme de terminaison de l'application

Comment on termine notre application

Variable : P → entier représentant les personnes qui n'ont pas atteint l'objectif.

Pré-condition : P > 0

REVENIR LA DESSUS <3

1) Avant chaque phase de déplacement, on vérifie si P > 0
2) Si une personne a atteint l'objectif alors P = P – 1
3) Sinon déplacement d'une personne.
4) Si toute les personnes ont atteint l'objectif alors fin du programme.

Version adaptée

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

int nbpeople , nombre de personnes sur la grille

```
for k allant de 0 à nbpeople
        si population[k].state == AVAILABLE
             execute_thread_movement() ;
        end
end
terminate_program() ;
```

6. Conclusion

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...))

+En java, pour chaque objet est déclaré un verrou (peut être utile ?)