Groupe 22 – Andolerzak – Ghira - Pépin

Introduction

Notre équipe a pour mission de représenter informatiquement le déplacement d'une foule à l'intérieur d'un terrain d'une taille de 512 x 128 pixels. Ce terrain comporte des obstacles (murs) et les personnes à l'intérieur ont une taille de 4 x 4 pixels et doivent se déplacer jusqu'à un point d'arrivée, sans collisions.

Pour se faire, nous devons utiliser les connaissances acquises lors des cours et TD de la matière programmation concurrentielle. Ce document présentera nos décisions ainsi que les justifications et techniques mises en œuvre afin de mener à bien notre projet.

Nous verrons aussi les algorithmes utilisés pour le déplacement d'un personnage ou le choix d'une direction par exemple.

Gestion des paramètres de lancement

Le programme doit gérer trois paramètres de lancement, qui sont **–p –t** et **–m**. Les deux premiers paramètres prennent en argument une valeur. Il faut donc vérifier que les valeurs entrées respectent les spécifications. C'est pour cela que notre objet **OptionChecker** va servir de point d'entrée de notre programme. Grâce à la fonction **getopt()** nous pouvons récupérer les éléments donnés lors du lancement du programme.

Création du terrain et des obstacles

Le terrain est créé par un **vector** de **vector** de struct_mutex. Cette structure contient un int qui indique si la case est occupée, et un sémaphore. Nous avons choisi des int dans la mesure où une case peut être occupée par un mur ou par une personne ou encore être libre et donc on les différencie avec un int dans la mesure où un bool ne pourrait pas répondre à ce besoin. Le sémaphore est présent dans le cas où nous choisissons d'exécuter la simulation avec la phase 2 et l'option t2, où l'intérêt est d'avoir un sémaphore pour chaque case. L'intérêt d'utiliser un vecteur est que l'on n'a pas besoin de connaitre précisément la taille de ton tableau au moment où te le crée, cela permet, si on le souhaite d'adapter la taille de la carte simplement sans faire de realloc ().

Réalisation des murs sur le terrain avec la méthode addWall() :

```
POUR I allant de 0 à la taille hauteur HEIGHT de la carte
        POUR J allant de 0 + la largeur du mur WIDTHWALL
                SI i correspond au mur le plus à gauche x de 0 à 15 et y de 0 à 59 et de 67 à 127
                        On met les cases de la grille à 1 correspondant à un mur
                FIN SI
        FIN POUR

        POUR J allant de 112 + la largeur du mur WIDTHWALL
                SI i correspond au mur le plus du milieu x de 112 à 127 et y de 0 à 55 et de 72 à 127
                        On met les cases de la grille à 1 correspondant à un mur
                FIN SI
        FIN
```

```
FIN POUR
```

Figure 1 : Algorithme de création de murs

Création d'un personnage

Un personnage représenté par l'objet **Character** possède plusieurs caractéristiques comme sa hauteur et sa largeur qui sont des valeurs fixes de 4 x 4, mais aussi sa position, qui est modélisée grâce à un objet **Point** défini par nous-même. Cet objet est un couple de coordonnées avec diverses opérations dessus dont notamment des accesseurs.

Notre moteur (objet **Motor**), qui sert d'intelligence dans notre programme, contient une liste de personnes. Afin de positionner les personnages sur le plateau, nous allons d'abord créer le nombre d'entités sans leur attribuer une position. Ce nombre est de 2^n, où n correspond à l'argument passé en paramètre lors de l'exécution du programme.

Déplacement d'un personnage

En fonction du scénario choisi, nous avons soit un **Thread** par personnage, soit quart de terrain, soit pour tous les personnages. En fonction de ça, le **Thread** courant s'occupant du déplacement d'un personnage va appeler la fonction ***moveAll** qui va parcourir la liste des personnages, et pour chaque personnage, récupérer sa position grâce à l'accesseur **getPoint()** de la classe **Point**, et appeler la fonction **changePosition()**. La méthode reçoit en paramètre les coordonnées d'une personne (son x et y), elle regarde si le point se trouve sur la partie :

On INITIALISE 3 int pour chaque déplacement à 0

```
SI (Y < 60 ET X<128) OU (Y<56 ET X>=128)
        POUR I allant de 0 à 5
                SI une case n'est pas libre
                        Si la case bloque déplacement Horizontal
                                LockH à 1
                        FI SI
                        Si la case bloque déplacement Diagonal
                                LockD à 1
                        FI SI
        FIN POUR

        POUR I allant de 0 à 4
                SI une case n'est pas libre
                        Si la case bloque déplacement Vertical
                                LockV à 1
                        FI SI
                        Si la case bloque déplacement Diagonal
                                LockD à 1
                        FI SI
        FIN POUR

        SI LockD a 0            retourne point Diagonal         FIN SI
        SINON SI LockH a 0      retourne point Horizontal       FIN SI
        SINON SI LockV a 0      retourne point Vertical         FIN SI
        SINON Aucun possible    retourne point actuel           FIN SINON
FIN SI
```

```
SINON SI (Y >= 65 ET X<128) OU (Y>=69 ET X>=128)
        POUR I allant de 0 à 5
                SI une case n'est pas libre
                        Si la case bloque déplacement Horizontal
                                LockH à 1
                        FI SI
                        Si la case bloque déplacement Diagonal
                                LockD à 1
                        FI SI
        FIN POUR

        POUR I allant de 0 à 4
                SI une case n'est pas libre
                        Si la case bloque déplacement Vertical
                                LockV à 1
                        FI SI
                        Si la case bloque déplacement Diagonal
                                LockD à 1
                        FI SI
        FIN POUR

        SI LockD a 0                retourne point Diagonal             FIN SI
        SINON SI LockH a 0          retourne point Horizontal           FIN SI
        SINON SI LockV a 0          retourne point Vertical             FIN SI
        SINON Aucun possible        ne retourne point actuel            FIN SINON
FIN SI

SINON
        POUR I allant de 0 à 5
                SI une case n'est pas libre
                        Si la case bloque déplacement Horizontal
                                LockH à 1
                        FI SI
        FIN POUR
        SI LockH a 0                retourne point Horizontal       FIN SI
        SINON Aucun possible        retourne point actuel           FIN SINON
FIN SINON
```

Figure 2 : Algorithme de déplacement d'un personnage

Nous avons décidé dans notre stratégie que si un joueur doit, par exemple, se déplacer dans une diagonale et qu'une des cases où le joueur doit se retrouver, est occupée, alors on regarde une pour un déplacement horizontal et s'il y a le même problème on essaye le déplacement vertical ; si le problème est encore présent, le joueur attend le tour d'après pour refaire ce même mouvement. Une autre stratégie aurait été de simuler ce mouvement avec un mouvement vertical puis horizontal (ou inversement), mais nous avons décidé de ne pas utiliser cette stratégie.

Gestion des threads

Le programme peut être lancé avec trois arguments dans le paramètre **–t.** C'est dans le moteur que l'on gère les différentes façons d'exécuter le programme en fonction de la valeur reçue (après avoir été vérifiée au préalable comme expliqué avant).

```
SI argument = 0
   CRÉER un thread.
   DEPLACER personnages
FIN SI
SI argument = 1
   CRÉER thread NORD-EST
   CRÉER thread SUD-EST
   CRÉER thread NORD-OUEST
   CRÉER thread SUD-OUES
   POUR TOUT personnage DANS NORD-OUES
      DEPLACER personnage avec thread
      ….
FIN SI
SI argument = 2
   POUR TOUT personnage
      CRÉER thread
      DEPLACER personnage
   FIN POUR TOUT
   ATTENDRE fin thread
FIN SI
```

Figure 3 : Algorithme de création de threads

Lorsque cette variable est à 0, le moteur appelle la fonction **moveAll()** qui s'occupe de faire bouger tous les joueurs jusqu'à leur arrivée au point **d'Azimuth** (point d'arrivée des joueurs). Cette fonction ne créé pas de thread car elle utilise le principal, qui est le même que le **Main**.

Lorsque la valeur est de 1, 4 threads sont créés, un pour chaque partie du terrain. Nous avons donc en tout 5 threads si on compte celui du **Main**. Pour chacun des threads créés, on lui passe en paramètre une fonction qui s'occupera de déplacer les joueurs. Le code correspondant est le suivant :

```
pthread_create(&t0, NULL, moveNO, this) ; //s'occupe de la partie nord-ouest

pthread_create(&t1, NULL, moveNE, this); //s'occupe de la partie nord-est

pthread_create(&t2, NULL, moveSO, this); //s'occupe de la partie sud-ouest

pthread_create(&t3, NULL, moveSE, this); //s'occupe de la partie sud-est
```

Figure 4 : Code gérant la création d'un thread pour chaque quart de terrain

Lorsque la valeur est de 2, il faudra créer un thread pour chaque personnage, le moteur s'occupe aussi de cela. Une boucle va être faite sur la liste des personnages afin de créer un thread à chaque fois. Ce thread va aussi être ajouté à un **vector** afin de permettre au processus père de pouvoir **join**, ce qui signifie qu'il va attendre la fin des processus. Lorsque la bouche permettant de créer le thread est faite, une autre boucle est créée afin de parcourir le vector de thread et faire le fameux join. La syntaxe est donc la suivante :

```
pthread_join(t0, NULL);
```

Figure 5: Code permettant au Main thread d'attendre une thread fille

Lorsque l'on créé le thread, nous lui passons en paramètre la fonction **movePerson()**, mais aussi la référence vers **data**, qui est une structure créée qui contient le moteur de jeu, un personnage, et son indice dans la liste des personnages.

La fonction **movePerson()** s'occuper de récupérer le personnage et va le déplacer jusqu'à arriver à destination. Lorsque le joueur est arrivé, on le retire de la liste.

Comparaison thread Java & Posix

Java	Posix
- créer un objet Thread (new thread) - lancer un thread Java, appeler la méthode start() qui appelle la méthode run() afin de lancer l'exécution de notre programme ou - créer un objet qui implémente l'interface **Runnable** et surchargera la méthode **run**. Afin d'exécuter le traitement de notre thread, il faut créer un objet Thread et passer en paramètre du constructeur, une instance de l'objet implémentant Runnable. Suite à la création de l'objet, on appelle, comme précédemment, la méthode start. - on peut demander au thread principale d'attendre la fin du thread fille - détruit lorsqu'il a fini d'exécuter la fonction donnée à sa création depuis la version 1.1 (la méthode **stop()** étant dépréciée	- créer en appelant une méthode (pthread_create()) avec en paramètre la fonction à exécuter. - la création du thread suffit à lancer le thread - on peut demander au thread principale d'attendre la fin du thread fille - détruit lorsqu'il a fini d'exécuter les actions de la fonction donnée en paramètre lors de sa création ou - pthread_exit

Comparaison sémaphore Java & Posix

Java	Posix
- créer un nouvel objet sémaphore - acquire permet de prendre la main, cette méthode est bloquante	- sémaphore nommé, Deux processus peuvent opérer sur le même sémaphore nommé en passant le même nom à sem_open

-tryAcquire demande une ou plusieurs autorisations - release rend une autorisation, ou le nombre d'autorisations passées en paramètre - availablePermits retourne le nombre d'autorisations disponibles pour ce sémaphore - assez proche du synchrinized mais avec le pouvoir gérer l'accès de plusieurs tâches	- sem_open crée un nouveau sémaphore nommé ou en ouvre un existant - sem_post et sem_wait permettent de travailler sur le sémaphore nommé -sem_close permet de fermer le sémaphore -sem_unlink permet de supprimer le sémaphore du système après la fin de tous les procéssus - sémaphore non nommé est placé dans une région de la mémoire qui est partagée entre plusieurs threads ou processus - sem_init initialise un nouveau sémaphore non nommé - sem_post et sem_wait permettent de travailler sur le sémaphore non nommé -sem_destroy permet de détruire un semaphores non nommé

Mesure des performances

Si l'utilisateur indique le paramètre **–m** lors de l'exécution, le programme doit être capable de mesurer la consommation du CPU, mais aussi le temps d'exécution du programme entre le lancement et la fin de la simulation. Afin de rendre nos mesures un peu plus précises, nous réalisons une moyenne des résultats sur 3 simulations sur 5, les deux retirés sont les extremums des mesures.

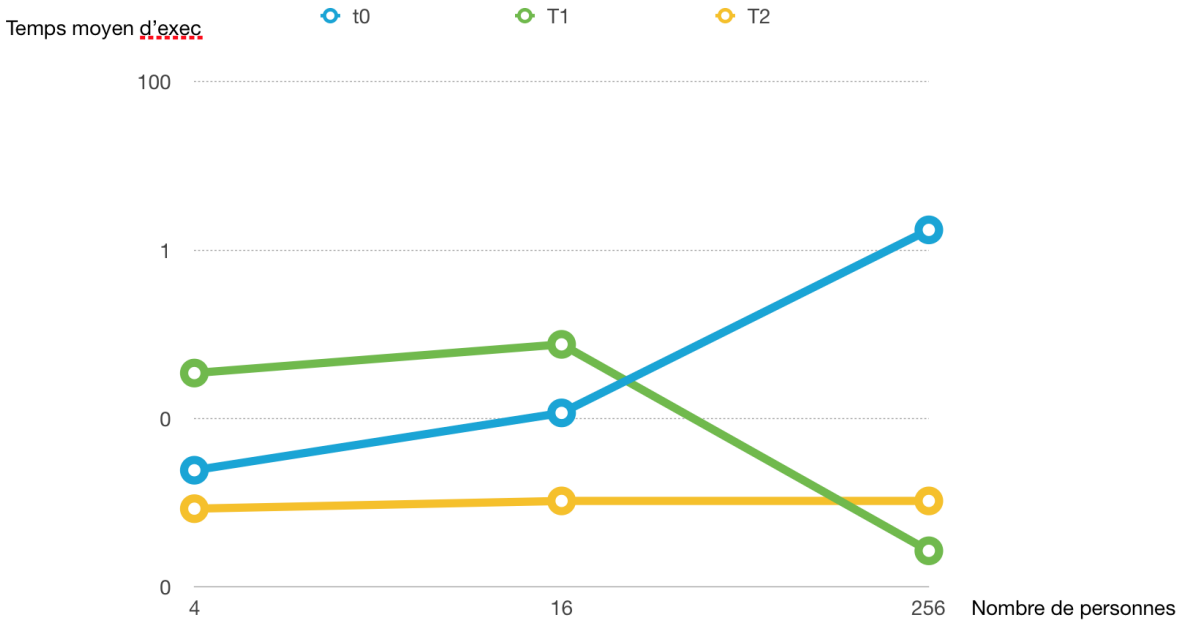
Pour mesurer le temps d'exécution nous avons récupéré l'heure en secondes avant le traitement des paramètres d'exécutions, et à la fin de notre programme. Une simple soustraction de ces deux valeurs grâce à **difftime()** nous permet de connaitre le temps d'exécution. Pour cela, il existe la fonction **time()** qui prend en paramètre une variable de type **time_t**.

Pour le temps CPU consommé, la fonction **clock()** est utilisée. Le principe est le même, nous prenons deux valeurs, une au début, et une à la fin pour obtenir le temps CPU consommé, exprimé en **CLOCKS_PER_SECs / s**. Nous divisons le résultat par **CLOCKS_PER_SECS** pour obtenir le temps en secondes.

Valeur pour e1 :

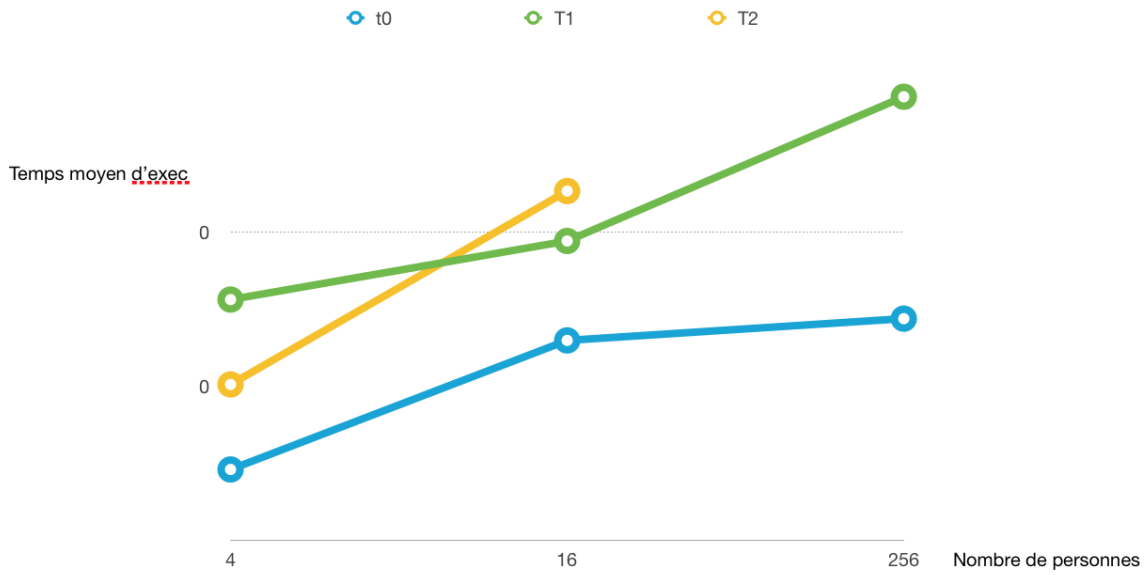
E1	T0	T1	T2
P2	Temps d'execution moyen ---> 0.002421. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.034594. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.000846. Temps de réponse moyen ---> 0.000000.
P4	Temps d'execution moyen ---> 0.011628. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.075880. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.001046. Temps de réponse moyen ---> 0.000000.
P8	Temps d'execution moyen ---> 1.739907. Temps de réponse moyen ---> 2.00746.	Temps d'execution moyen ---> 0.000267. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.001046. Temps de réponse moyen ---> 0.000000.

Figure 6 : Mesures de différents scénarios d'exécution e1



Valeur pour e2 :

E2	T0	T1	T2
P2	Temps d'execution moyen ---> 0.000289. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0,036546. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0. 010290. Temps de réponse moyen ---> 0.000000.
P4	Temps d'execution moyen ---> 0.019858. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.087650. Temps de réponse moyen ---> 0.000000.	Temps d'execution moyen ---> 0.184781. Temps de réponse moyen ---> 0.000000.
P8	Temps d'execution moyen ---> 0. 027489. Temps de réponse moyen ---> 0.000000	Temps d'execution moyen ---> 0. 0,755326. Temps de réponse moyen ---> 1.027289.	INFINI



Les seules conclusions que nous pouvons avoir sont sur le scenario avec t0. Nous remarquons que le temps CPU augmente en fonction du nombre de personnes. En effet, seul un thread doit gérer les déplacements des personnages, c'est pour cela que le temps d'exécution passe de 0 seconde à 1 seconde entre 4 personnes et 256 personnes. L'augmentation est aussi présente pour le temps CPU car on sollicite plus longtemps le CPU à cause du fait que nous avons qu'un seul thread.

Concernant le couple (t2 ; P2), nous voyons un temps CPU plus bas par rapport au couple (t0 ; P2), ce qui montre que l'utilisation de plusieurs threads utilise moins longtemps le CPU.

Pour les valeurs manquantes, cela est, selon nous dû à un problème de performance. Lorsque nous lançons ces simulations sans le paramètre **–m**, le programme s'exécute assez rapidement. Dès lors que nous rajouter les mesures, le programme ne termine pas. Après avoir vérifié que ce n'était pas un problème de code, nous avons remarqué que lors de l'exécution, tous les cœurs de notre processeur étaient à 100%. N'ayant pas de machines plus performantes que celles actuelles (processeur i5), nous n'avons pas pu cerner si le problème était dû à une non-optimisation du programme ou de paramètres d'exécution trop gourmands. C'est une piste d'amélioration de notre programme pour les prochaines étapes.

Analyse des scenarios

L'ajout de sémaphores lors de la phase 2 assure une cohérence dans les déplacements ainsi que les résultats. En effet lors de la première phase, on pouvait se retrouver avec des personnes qui consultent une case libre et ensuite s'y déplacent en même temps. Maintenant que nous avons des sémaphores (ici des mutex), nous sommes assurés qu'un personnage se déplacera sur une case que si le mutex n'est pas bloqué.

Une façon de modéliser ça en FSP serait de cette façon :

```
INITIALISATION :
        POUR TOUT case
        CRÉER SEMAPHORE
DEPLACEMENT :
        SI  pas de mutex aux alentous
                verrouiller mutex
        SINON
                ATTENDRE
```

Figure 7 : Pseudo modélisation FSP

Conclusion

En conclusion, nous pouvons retenir qu'actuellement, nous sommes capables de faire travailler des threads en parallèle afin de terminer plus rapidement notre programme Cela résulte par plus d'un personnage présent sur les mêmes cases.

Cependant, l'utilisation actuelle des mutex pourrait poser problème car il est probable que deux processus ou plus entrent en inter blocage, et de ce fait le programme ne s'arrête pas.

Nous avons aussi constaté dans l'équipe que la gestion des threads en Posix est plus pratique qu'en Java car nous n'avons pas à implémenter une interface ou créer un objet. Il suffit juste de créer le thread et stipuler la méthode à traiter, nous définissons ainsi explicitement son cycle de vie.

Enfin, les mesures prises nous ont permis de voir les limites de notre programme. Comme expliqué précédemment, nous devons cibler le problème et comprendre la cause de notre programme qui ne s'arrête pas.