Informatique embarquée : TP n°2

Temps estimé : ~ 2 heures 30.

Temps passé : ~ 3 heures.

Remarques

J'ai perdu du temps sur la modification de mon *Makefile*. C'était dû à un manque d'organisation. J'ai pu refaire les choses tranquillement à tête reposée.

De plus, j'ai résolu mon problème à propos de l'option *-m32* sur ma machine. Il fallait juste installer un paquet et le problème était réglé.

Pour revenir sur le TP, j'ai fait un fichier *tp2.c* qui contient la fonction principale utilisant la bibliothèque statique *collimacon.a*. J'ai également défini la règle pour compiler vers la bibliothèque dynamique *collimacon.so*.

En ce qui concerne le programme de test automatisé, j'ai crée un progrmma en C qui contient les tableaux résultats, lance la fonction *collimacon()* qui génère le tableau et compare le tableau généré avec le tableau attendu. Il fait les tests pour différents cas.

Tous les tests ont été fait avec le programme utilisant la bibliothèque statique.

À propos du prototype

Dans le cadre de l'établissement de la bibliothèque *collimacon*, j'ai défini le prototype suivant :

```
int32_t * collimacon(int32_t width, int32_t height)
```

Ici, la fonction *collimacon()* va créer le tableau, le remplir, et l'envoyer en retour à la fonction appelante, qui sera responsable de la libération mémoire. L'avantage, dans l'usage de cette fonction, est que la condition de succès ou d'échec sera déterminée par l'état du tableau (alloué ou non). Si le tableau a été alloué sans problème, alors il a été rempli selon l'algorithme « collimacon », sinon le tableau vaut NULL.

Le prototype suivant :

```
int collimacon(int ** tab, int width, int height)
```

pose problème dans la mesure où il faudrait fournir un tableau (alloué ou non). Il faudrait donc vérifier au préalable si le tableau a été alloué. Dans le premier prototype on n'a pas ce problème.

Parrallélisme

En l'état, le programme ne serait pas parallélisable sans modification importante. Cependant, on pourrait définir un algorithme concurrent avec 4 threads (un thread par côté), à condition que l'on spécifie correctement les positions de départ et d'arrivé.

En revanche il serait trop compliqué de paralléliser le remplissage de manière à avoir un thread par processus. En effet, il faudrait prendre en compte l'ordre de parcours et connaître la valeur de la case précédemment remplie. Cela impliquerait qu'un thread ne pourra remplir la case correctement que si ses prédécesseurs l'on fait (sauf pour le premier thread). De plus, d'un point de vue technique, il ne serait pas souhaitable d'adopter ce genre de solution dans la mesure ou sur un tableau de *100 000 x 100 000* cases, on aurait *100 000 000* threads. On aura donc des problèmes de ressources. C'est particulièrement vrai sur des systèmes embarqués.

Temps d'exécution

Dimension	Temps (en ms)
1 x 1	~ 1
10 x 10	~ 1
100 x 100	~ 1
1000 x 1000	~ 15
10000 x 10000	~2100

Ces calculs ont été effectués avec la commande *perf stat*.

Accès au tableau

Pour un tableau de **N** cases, il faudra **N** accès, en lecture/écriture.