

Documentation multi-threading library

By mdarty – V0.1.3

Table des matières

| | |
|--|----|
| Introduction..... | 2 |
| Présentation de la librairie..... | 2 |
| Fonctionnement de la librairie..... | 3 |
| Comment penser votre programme avec mtlb ?..... | 4 |
| Fonctionnalités..... | 8 |
| A : Prérequis d'utilisation de la librairie..... | 8 |
| B : Initialisation et nettoyage..... | 8 |
| C : Envoi de fonction à la librairie..... | 9 |
| D : synchroniseur de fonctions..... | 11 |
| E : Fonction d'exécution de fonction..... | 12 |
| F : Création/destruction de thread..... | 13 |
| Démonstration..... | 14 |
| A : Sans ma librairie :..... | 15 |
| B : Avec mtlb :..... | 16 |
| Licences des sources :..... | 19 |
| Remerciement à :..... | 20 |

Introduction

Présentation de la librairie

Cette librairie est une surcouche de la librairie pthread, développée en C, utilisable en C/C++ et est compatible avec les systèmes d'exploitations UNIX/Linux, Mac OS X et Windows. Celle ci permet d'utiliser, nativement et sans risque de blocage, tous les cœurs de votre processeur sans se préoccuper de la création, destruction, répartition des actions sur les différents threads.

Pour les puristes cherchant la performance et éviter certains goulet d'étranglement, cette librairie permet également d'attribuer des actions spécifiquement à un thread choisi.

Fini le temps où vous deviez penser à créer des variables d'exclusion mutuelle (mutex) et les déverrouiller, avec des risques de blocages infini. Cette librairie s'en occupe pour vous. La programmation multithread sera à présent beaucoup plus simple en utilisant cette librairie.

Je tiens à préciser que cette librairie vous poussera à penser différemment vos programmes. Pour l'utiliser de façon optimale, il faudra vous poser les questions suivantes :

- Quel sont les blocs d'actions qui peuvent s'exécuter en parallèle ?
- A quel moment la fin des actions doit être synchronisé ?

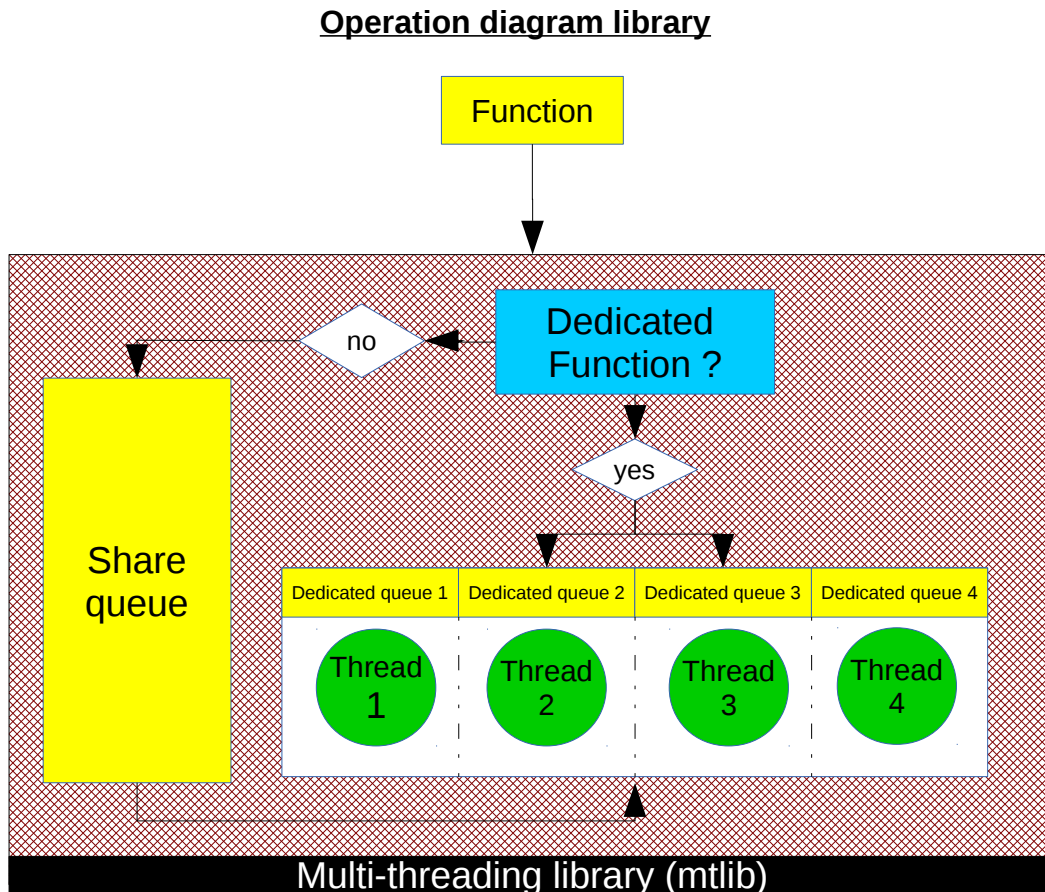
Tous les outils pour répondre à ces deux questions vous sont offerts dans cette librairie. Pour utiliser correctement cette librairie, je vous recommande vivement de maîtriser correctement les pointeurs.

Le dépôt contenant les sources de cette librairie est le suivant :

<https://github.com/darkael88/mtlib>

Fonctionnement de la librairie

Pour ceux qui se demandent comment j'ai conçu, voici ci-dessous un schéma qui présente sa conception. Pardonnez-moi le choix des couleurs pas très esthétiques.



Autor : Michaël DARTY

L'organisation est basée sur le principe d'un donneur d'ordre principal (main core) et de ses sous-traitants (les threads).

Pour schématiser, chaque fonction est envoyée dans le moteur. Deux possibilités :

- soit le développeur lui a spécifié un thread dédié, et dans ce cas elle est ajoutée au fil dédié du thread choisi.

- soit le développeur a laissé le choix au moteur pour la répartition. Cette fonction sera mise à la fin de la file partagée. Le moteur répartira les fonctions de cette file d'attente en les distribuant sur les threads libres selon la méthode FIFO ou LIFO. Le fil d'action dédié est prioritaire à la file d'action partagée.

A noter que les threads eux-mêmes peuvent sous-traiter des actions au moteur de la librairie qui sera sous-traité : soit à un thread dédié soit mis sur la file partagée. Le cœur principal peut aussi s'exécuter lui-même les actions si le développeur le souhaite.

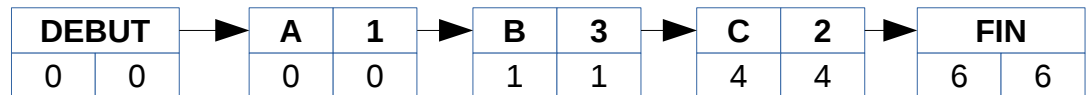
Comment penser votre programme avec mtlib ?

Lorsque vous développez en C, vous pensez les différentes tâches qui s'exécuteront dans un ordre prédéterminé. Le schéma pourrait être le suivant :

Ordonnancement des fonctions dans un programme C standard

Légende :

| Nom Fonction | Durée fonction |
|---------------------|----------------------|
| Date au Plus tôt | Date au Plus tard |



Tout d'abord le programme démarre, ce qui correspond à l'action « DEBUT ». Il exécutera l'action A qui a une durée de 1. Quand cette tâche sera finie, il exécutera la tâche B pour une durée de 3 ...

A la fin, la durée dans l'exemple du schéma ci-dessus sera de 6.

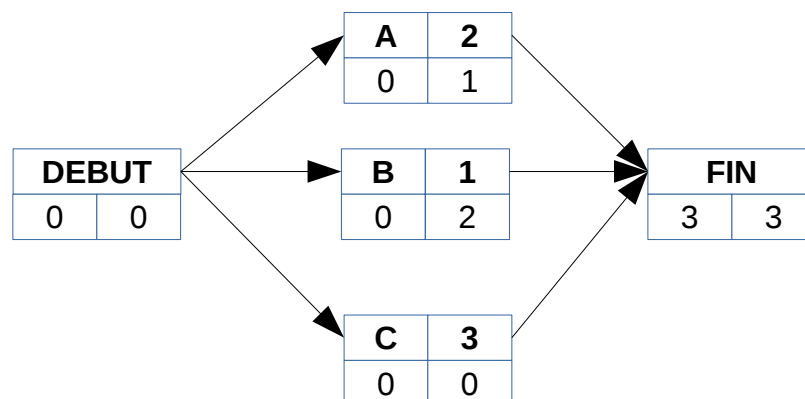
Dans les programmes par défaut sans utilisation des threads, les tâches s'exécuteront une par une par une seule ressource : un coeur. Mais si notre processeur possède plusieurs coeurs, que se passe-t-il ? Ben rien, les autres coeurs ne sont pas utilisés dans notre programme ci-dessus.

Si on utiliserait tous nos coeurs de notre machines, le schéma sera le suivant :

Ordonnancement des fonctions dans un programme C multithread

Légende :

| Nom Fonction | Durée fonction |
|---------------------|----------------------|
| Date au Plus tôt | Date au Plus tard |



Toutes les tâches n'ayant aucune dépendances entre elles, peuvent s'exécuter en parallèle, ce qui

réduit le temps d'exécution total du programme à 3 (contre 6 précédent).

La librairie présenté dans ce document permet de développer exactement comme le schéma ci-dessus, c'est à dire en parallélisant les tâches. Les actions seront attribuées au premier coeur disponible pour utiliser au mieux les ressources matériels à notre disposition. Cette librairie vous oblige donc d'avoir un raisonnement de gestion de projet.

C'était un exemple simple.

Maintenant imaginons le programme suivant :

Un programme à 10 fonctions à lancer. Ces fonctions ont les dépendances suivantes, c'est à dire ne peut s'exécuter que lorsque une action précise soit terminé :

A : Aucune

B : aucune

C : A

D : Aucune

E : C

F : D

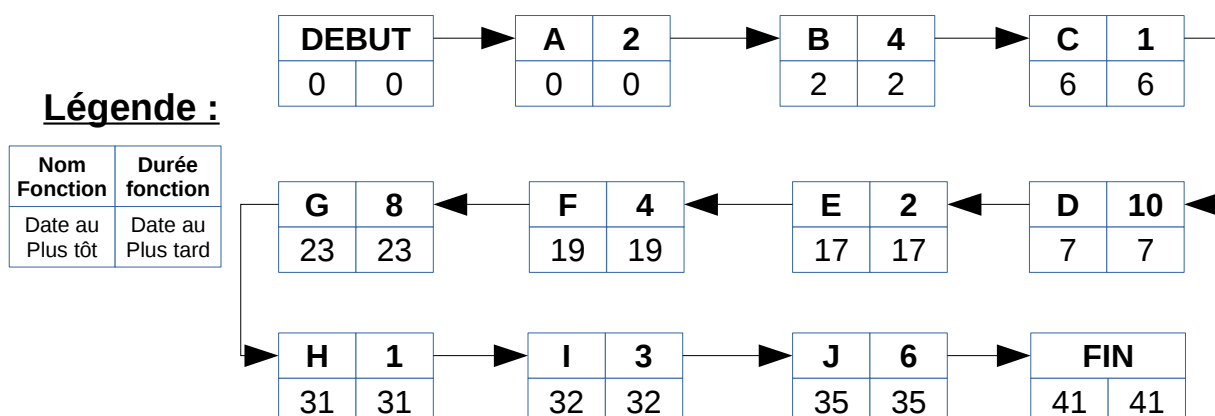
G : B

H : E, F

I : H

J : G, I

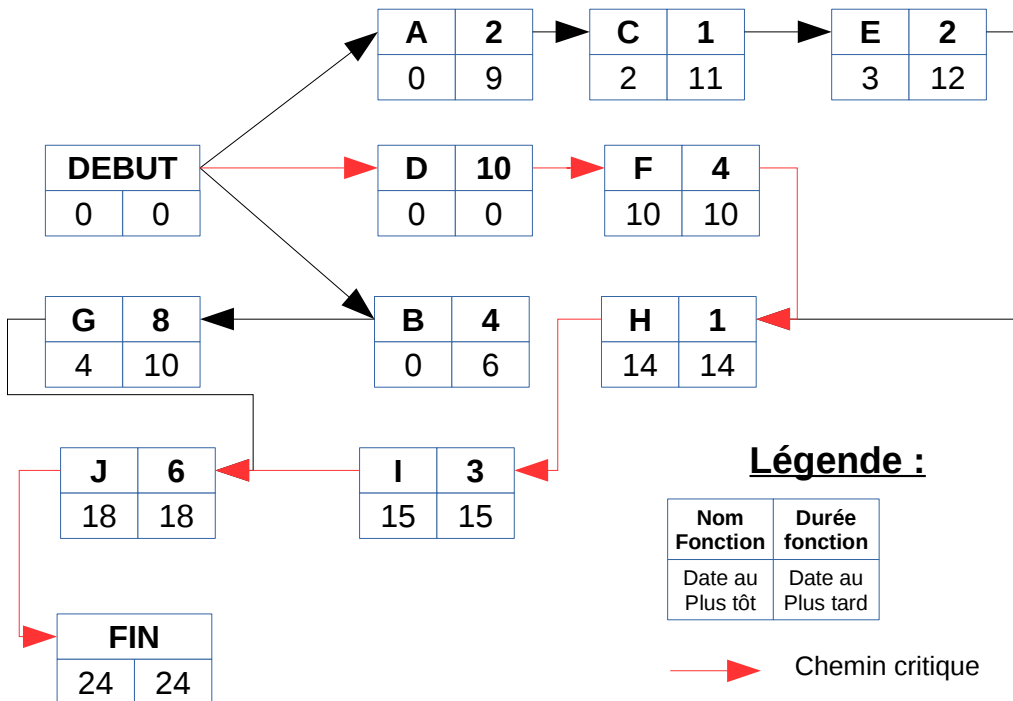
Dans un programme classique en C, on se contentera d'exécuter les fonctions une à une comme le schéma suivant :



Comme vous pouvez voir, le programme n'est pas optimale dans la parallélisation des tâches. Pourtant, c'est ce que fait lorsqu'on programme en C de manière classique.

En programmant en pensant multithread et avec la mtlib, le schéma optimal du programme sera le suivant :

Ordonnancement des actions optimale du programme :



Dans le schéma ci-dessus, le temps optimal d'exécution du programme est de 24 (contre 41 précédemment) et le maximum de ressources utilisés simultanément est de 3 cœurs,

Je sais ce que vous vous dites, c'est bien beau tous ces schémas, mais comment ça s'illustre concrètement avec la librairie de ce document ?

Pour gérer les dépendances, les différentes tâches doivent être appelé depuis une fonction mère qui contiendra les fonctions contenant les dépendances. que le développeur définira. Exemple :

-Au niveau de début, il faudra initialiser le moteur via la fonction [mtl_init](#) et au niveau de la fin, il faudra appelé la fonction [mtl_end](#) qui nettoiera le tout.

-Les fonctions A, D et B doivent être appelé depuis la même fonction qui a été envoyé au moteur de la librairie. La fonction d'envoi dans le moteur est : [mtl_send](#), la fonction mère peut elle même être envoyé au moteur.

-Pour synchroniser les dépendances, par exemple au niveau de la fonction H, qui héritera du résultat de E et F, il faudra utiliser utiliser la fonction [mtl_join_status](#) qui attendra que toutes les fonctions dont elles héritent soient terminé. La fonction H doit être appelé idéalement au niveau de la F et doit avoir l'adresse du status de A. H sera exécuté quand E sera terminé.

Pour schématiser, la fonction mère pourrait être la suivante :

```

void                function_mother(void)
{
    int             status[3] = {0} ;//permet de connaître la fin de la fonction
    mtl_join        *join ;
    void            *destA ;//resultat de la fonction A
    void            *destB ;
    void            *destC ;

    /* initialisation du bloqueur et envoies des fonctions A, B et D */
    mtl_send(STD, &status[0], &destA, function_A, 0) ;
    /*
    ** la fonction A executera aussi C et E , le status sera modifié uniquement quand E sera
    **terminé
    */

    mtl_send(STD, &status[1], &destB, function_B) ;
    /*au sein de ce bloc, le status sera mis sur 1 quand G sera terminé */

    join = mtl_init_join(join, &status[2]) ;
    mtl_send(STD, &status[1], &destD, function_D, 2, &status[0], &status[1]) ;
    /*au sein de ce bloc, il devra attendre la fin de A au niveau de E et B au niveau de G
    le status sera mis à terminé seulement à ce moment là
    */

    mtl_join(join, STD) ;// attend que I et G soit terminé.

    Function_J() ;//execute seulement maintenant a fonction J
}

```

Cela vous semblera probablement difficile à appréhendez, je ne vous ai mis uniquement la fonction au niveau qui est exécuté juste avant l'appelle de A, B e D. Je vous recommande donc vivement de vous entraîner à faire des programmes en vous habituant à découper vos tâches et vous poser principalement la question suivante : Est ce que cette action nécessite le résultat d'une autre fonction ? Si oui, laquelle ?

Fonctionnalités

A : Prérequis d'utilisation de la librairie

Pour utiliser cette librairie, vous devez ajouter la ligne suivante dans chacun de vos fichiers l'utilisant :

```
#include <mtlib.h>
```

Pour compiler vos applications avec cette librairie, il vous faudra faire le lien avec la librairie **mtlib.a**, **mtlib.so** ou **mtlib.dll** avec l'option **-pthread** (ou plus tard, directement avec les sources).

Et voilà vous pourrez créer et compiler vos programmes en C/C++.

B : Initialisation et nettoyage

Initialisation :

Dans tous vos programmes, pour permettre le fonctionnement de cette librairie, vous devez appeler la fonction :

```
int mtl_init(const int nb_thread);
```

Cette fonction prend en paramètre le nombre de thread que vous souhaitez créer.

Quelque soit la valeur : le moteur sera tenté d'être initialisé.

Pour les valeurs suivantes :

nb_thread < 0 : Le nombre de thread adapté à votre machine sera créé. Pour simplifier la lecture de votre code, je vous recommande vivement d'utiliser la macro **NB_THREAD_STD**.

nb_thread >= 0 : Le nombre de thread que vous souhaitez avoir. A noter que 0 est une valeur valide, vous n'aurez juste aucun thread de créer mais le moteur sera initialisé.

L'appel de cette fonction retourne le nombre de thread créé et -1 en cas d'erreur.

Nettoyage

Quand vous ne souhaitez plus utiliser la librairie, afin de nettoyer proprement les éléments en mémoire (programmation en C/C++ oblige), il vous faudra utiliser la fonction suivante :

```
void mtl_end(void)
```

C'est tout, cette fonction fera le ménage pour vous toute seule et donnera l'ordre au thread de quitter dès qu'ils n'auront plus rien à faire. Vous n'avez pas besoin de gérer vos threads.

C : Envoi de fonction à la librairie

A l'étape précédente, vous avez initialisé le moteur, à présent, nous allons voir comment envoyer les fonctions au moteur. Autant vous prévenir, la fonction à utiliser vous semblera « horrible » au première abord mais nous allons la décortiquer ensemble.

Voici son prototype complet :

```
void mtl_send_function(int thread, int * status, int size_ret, int type_ret,  
void * dest, void *(*ft)(), unsigned int nb_param, ...);
```

Oui, je sais ça pique les yeux, mais dites vous que cette fonction fera (presque) tout pour vous. Cette fonction envoie votre fonction à la fin d'une file dédiée ou de la file partagé.

Pour « simplifier », vous pouvez utiliser la macro suivante (et je vous le recommande):

```
MTL_SEND(thread, status, type_ret, dest, fct, nb_param, ...)
```

Dans cette macro, vous n'avez pas besoin de préciser la taille de la destination ni de caster en (void *(*)(())) votre fonction, cette macro le fera pour vous. Je vous recommande, par mesure de simplification d'utiliser cette macro.

Si vous souhaitez utiliser une fonction qui ne renvoie rien ou que sa valeur de retour ne vous intéresse pas, vous pouvez utiliser la macro :

```
MTL_SEND_NO_RET(thread, status, fct, nb_param, ...)
```

Cette macro ne permet de saisir que le numéro du thread, le statut (que vous pouvez remplacer par la macro « NO_STATUS » également), la fonction à envoyer avec ces paramètres.

Mais nous allons quand même voir ces arguments un par un :

thread : On décide d'affecter ou non à un thread particulier. Les threads sont notés à partir du numéro 0. Si vous souhaitez que la fonction répartisse pour vous votre fonction, mettez la macro STD.

Status : Prend un paramètre l'adresse d'une variable ou stocké son statut en cours. Ce paramètre vous servira si vous souhaitez contrôler que la fonction c'est bien terminé.

Si vous souhaitez utiliser ce paramètre pour mettre un point d'arrêt dans votre programme en attendant la fin de réalisation de la fonction envoyé, initialisez la valeur pointée par statut à 0. La fonction, quand elle sera terminée, mettra la valeur sur 1

Pour les fonctions sans retour, la valeur à tester de statut doit être != à 0.

Size_ret : Calcul la taille de du type de la destination. Comme la destination est forcément une adresse, vous devez faire sizeof(*dest).

Type_ret : Indique si le type pointé par dest est un pointeur ou une valeur. Si c'est une valeur, il faut indiquer la «VALUE » sinon « PTR ». Les tableaux statiques sont des valeurs.

Dest : C'est l'adresse où sera envoyée la valeur de retour de la fonction envoyée en paramètre.

Si la fonction ne doit rien renvoyer, veuillez utiliser la macro « NO_RET » pour éviter le segfault (size_ret se sert de se paramètre pour calculer la taille de la destination).

Ft : On y est, c'est la fonction que vous voulez envoyer aux threads. Ce paramètre prend n'importe quel type de fonction. Attention, si vous souhaitez utilisé des types personnalisés, veuillez concevoir votre fonction en ayant en paramètre des pointeurs et non des valeurs.

nb_param : Vous devez préciser le nombre de paramètres que possède la fonction. La limite gérée est de 10 paramètres.

... : Vous devez mettre tous les paramètres que vous souhaitez envoyer à la fonction, que ça soit des valeurs (si c'est limité au type de base int, float, double, char ...) ou des pointeurs. Si vous ne mettez pas les bon paramètres, le comportement de la fonction sera indéfini.

Et voilà, vous savez tout ce qui est nécessaire pour utiliser la librairie, avec juste ces 3 fonctions, si celles-ci n'ont pas trop de dépendances entre elle, vous pourrez programmer de manière asynchrone.

Warning : Je recommande vivement, pour toutes les opérations d'entrée / sortie avec les fonctions comme read, scanf, printf, write ... de les spécialiser au sein d'un seul et même thread afin d'éviter

les pertes de performances. Je vous invite à rechercher sur google pourquoi;).

Variante :

Nous avons vu la fonction qui envoie en **fin** de file des actions. La même fonction existe si vous souhaitez qu'elle soit exécuter en priorité en les mettant en **début** de file. Cette fonction se nomme `mtl_priority_send_function` et les macros sont `MTL_PRIORITY_SEND` et `MTL_PRIORITY_SEND_NO_RET`. Elles prennent exactement les mêmes paramètres.

D : synchroniseur de fonctions

Il se peut que dans votre programme, vous ayez besoin d'attendre qu'un certain nombre de fonction se termine. Le paramètre « statut » de la partie précédente répond à cet objectif certes, mais que fera votre programme en attendant qu'il se termine ?

Les fonctions de cette partie répondent à cette problématique en permettant l'exécution de fonction depuis le main cœur/thread en attente d'un retour.

Les étapes sont les suivantes:

Initialisation de la liste de status d'attente

Avant toute chose, vous devez créer le synchroniseur à l'aide de la fonction suivante :

```
mtl_join *mtl_init_join(mtl_join *list, int *status);
```

Le premier paramètre prend soit une liste existante, et le statut sera ajouté à la liste de référence, soit une valeur NULL, et dans ce cas un maillon de liste sera créé.

Vous pouvez lier un nombre illimité (relativement à la mémoire) de fin de fonction entre elle. Elle renvoie un pointeur sur le début de la liste ou NULL en cas d'erreur.

Bloqueur

Cette fonction a pour objectif de mettre un point d'arrêt dans le programme en attendant que toutes les fonctions que vous souhaitez contrôler soit terminées (via le paramètre statut). Mais au lieu d'attendre, que toutes les fonctions se terminent, le programme va exécuter d'autres actions en file d'attente. Par défaut, cette fonction va exécuter uniquement celles présentes en file d'attente

partagée.

Le prototype de la fonction est le suivant :

```
void mtl_join_status(mtl_join *list, int num_thread)
```

Le premier paramètre correspond à la liste initialisée précédemment contenant les éléments bloquant. Le second paramètre peut avoir les valeurs suivantes :

SI < 0 , dans ce cas seul la file d'attente partagée sera analysée et exécutée.

Sinon, en priorité la file d'attente dédié au thread nommé sera analysée ou, en cas de file dédiée vide, une action de la file partagée.

Ce processus se reproduira tant que tous les éléments de la liste auront le statut terminé (échec ou succès).

Pour éviter les appels trop fréquent à malloc, un certain nombre d'éléments ne seront pas détruit après leurs utilisations pour être recyclés par la suite.

E : Fonction d'exécution de fonction

Dans certains, vous pouvez tout simplement avoir le besoin d'exécuter une fonction présente dans la file d'attente partagée et/ou dédiée de force.

Ces fonctions ont les prototypes suivantes :

```
int mtl_exec_one_function(int thread);
```

La fonction ci-dessus contrôle la liste d'attente dédié du numéro de thread indiqué en priorité et, si la liste d'attente est vide, une fonction de la file d'attente partagée.

Elle retourne 1 si cette fonction arrive à exécuter une fonction ou 0 en cas de file d'attente vide à ce moment. Si vous précisez un numéro de thread qui n'existe pas, vous allez probablement segfault.

Les variantes existent pour exécuter une seule fonction d'une liste dédiée ou de la file partagée. Ces fonctions retournent les mêmes valeurs. Les prototypes sont les suivants :

```
int mtl_exec_one_dedicated_function(int thread) ;  
int mtl_exec_one_share_function(void) ;
```

F : Création/destruction de thread

Entre la phase d'initialisation et de nettoyage, il est possible que vous souhaitez ajuster le nombre des threads en activité. Une fonction est faite pour cela. Attention : cette fonction doit être utilisé entre la fonction `init_mtl` et `end_mtl`.

```
unsigned int mtl_adjust_nb_thread(int nb_thread) ;
```

Cette fonction ajuste le nombre de thread à la hausse comme à la baisse.

Elle prend en paramètre le nombre de thread total souhaité. une valeur négative ajustera au nombre de thread standard.

Elle retourne le nombre de thread. Les threads détruits seront ceux ayant le numéro le plus élevé.

Voilà toutes les fonctions de la librairie. Je l'ai pensé pour qu'elle puisse répondre à un maximum de besoin de développement. Si vous voyez d'autre fonctionnalité à développer, merci de m'informer.

Démonstration

On va à présent exécuter en boucle une fonction qui va calculer et incrémenter un double.

La fonction de test est la suivante :

```
int          ft_test3(int i)
{
    double          j = 0;

    while (j < 500000)
    {
        j += 0.01;
    }
    fprintf(stdout, "t");
    return (i);
}
```

Les Macros suivantes seront définies :

NB = 100 ;

A : Sans ma librairie :

Voilà à quoi ressemble la fonction principale de base :

On enregistre la valeur de retour de `ft_test3` dans le tableau (qui correspond à `i`). Puis on affiche ce numéro.

```
int      main()
{
    unsigned int  i;
    int           value[NB] = {0};

    i = 0;
    while (i < NB)
    {
        value[i] = ft_test3(i);
        i++;
    }
    i = 0;
    while (i < NB)
    {
        printf("%d\n", value[i]);
        i++;
    }
    return (0);
}
```

B : Avec mtlib :

La machine sur laquelle j'effectue mes testes est une machine virtuelle auquel j'ai alloué 6 cœur.

Voici le main correspond à l'appelle des fonctions de la lib (En jaune, celles spécifiques à la librairie pour montrer les écarts avec la version original).

```
int          main()
{
    unsigned int  i;
    int           for_thread[NB];
    mtl_join      *join = NULL;

    mtl_init(STD);
    i = 0;
    while (i < NB)
    {
        MTL_SEND(STD, &status[i], &for_thread[i], ft_test3, 1, i);
        join = mtl_init_join(join, &status[i]);
        i++;
    }
    mtl_join_status(join, STD);
    mtl_end();
    i = 0 ;
    while (i < NB)
    {
        printf("%d\n", for_thread[i]);
        i++;
    }
    return (0);
}
```

J'ai besoin d'un tableau contenant les valeurs pour bloquer le programme (statut) ainsi qu'un pointeur sur le bloqueur que j'initialise à NULL. Le nombre de thread optimal sera créé via la macro STD.

Au lieu d'appeler ft_test3 directement, je l'envoie en fonction de répartition de thread.

Juste après, j'envoie l'élément créé à la fonction `mtl_init_join` et remplira une structure (**cette partie sera modifiée pour être optimisée**) qui me servira de bloqueur en sortant de la boucle via `mtl_join_status`.

Juste après, comme je ne compte plus utiliser les threads, je fais le ménage.

Voici la comparaison entre les deux versions en terme de rapidité :

Sans mtlib

```
97
98
99

real      0m9.496s
user      0m9.500s
sys       0m0.000s
darkael@darkael-lmde:~
```

Avec mtlib (avec paramètres STD)

```
97
98
99



real      0m1.733s
user      0m9.928s
sys       0m0.200s
darkael@darkael-lmde:~/testc/m
```

Avec ma lib, dans le cas présent, 5 threads ont été créés adaptés à ma machine 6 cœurs (Main core + 5 threads) avec les paramètres standards.

Le temps d'exécution a été presque divisé par 6.

Dans le moniteur système, nous voyons bien que le CPU est utilisé au maximum (600%) et la consommation mémoire de la librairie est marginale.

Charge système pour les 1, 5, 15 dernières minutes : 3,45; 1,63; 0,77

| Nom du processus | État ▼ | % CPU | Priorité | ID | Mémoire | Canal d'attente |
|---|----------|-------|----------|-------|-----------|-----------------|
|  test_mtlib | En cours | 600 | 0 | 38004 | 348,0 Kio | 0 |
|  mate-system-monitor | En cours | 6 | 0 | 37924 | 13,8 Mio | 0 |

Pour toute remarque, ajout de fonctionnalité, critique, n'hésitez pas;). Et éventuellement, m'indiquer toutes les fautes d'orthographe qui doivent être légion.

Licences des sources :

La licence appliquée par défaut est CeCILL , compatible avec la GNU GPL et au droit français. Les obligations sont décrites dans le document LICENCE-EN (version écrite en anglaise) ou LICENCE-FR (Version écrite en français) à la racine du dépôt <https://github.com/darkael88/mtlib>

Néanmoins, si vous souhaitez une licence plus permissive de cette librairie, merci d'effectuer votre demande à mdarty.mtlib@gmail.com afin que je puisse l'examiner.

Remerciement à :

Ludovic RENAUDIN pour la relecture de ce document.