Ecole Nationale Supérieur d'Informatique pour l'Industrie et l'Entreprise (ENSIIE)

Programmation à base de thread
Création d'une bibliothèque de thread : *mthread*

Etudiant :
Romain PEREIRA

Encadrants :
Marc PERACHE
Arthur LOUSSERT

10 mars 2019

Résumé

Dans le cadre de la formation CIDM (Calcul Intensif et Données Massives) à l'ENSIIE, j'ai étudié la programmation à base de threads, puis j'ai implementé une bibliothèque de thread POSIX : mthread.

Cette implémentation a été guidé, et réalisé au cours de plusieurs séances de TP.

Ce rapport détaille l'implémentation des différentes fonctionnalités de la bibliothèque, en expliquant en profondeur son fonctionnement.

Le **sujet et le projet de départ** sont disponibles à l'adresse :
http://skutnik.iiens.net/cours/2A/PBT/TD2_mthread/

La **version finale de mon implémentation** est disponible à l'adresse :
https://github.com/rpereira-dev/ENSIIE/tree/master/UE/S4/PBT/mthread

Table des matières

Chapitre 1

A propos du rendu

Architecture du rendu
Le rendu suit l'architecture du projet de départ qui nous a été fourni.

Les tests
Les tests sont situés dans le dossier **tests** à la racine du projet.
Chaque test a le format suivant :
— Ecrit dans un fichier ***nom*.c**
— Compilé vers un programme ***nom***
— Chaque test est commenté en début de fichier source, expliquant la fonctionnalité testé, et le résultat attendu.
Pour compiler tous les tests, il suffit de taper la commande :

```
> make
```

Pour compiler et lancer tous les tests, il suffit de taper la commande :

```
> make test
```

Pour lancer les tests un à un :

```
> make [TEST]
> ./[TEST]
```

Chapitre 2

Découverte de la bibliothèque mthread

2.1 L'ordonnanceur

2.1.1 Explication du code de l'ordonnanceur (2.1 et 2.2)

Le code de l'ordonnanceur se situe dans la fonction *mthread_yield()* du fichier mthread.c.

Ce qui est appelé *le thread* ***idle*** correspond à un thread qui ne fait rien : il boucle indéfinément sur l'ordonnanceur, jusqu'à ce qu'un ordre thread prenne la main.

L'ordonnanceur fonctionne de la manière suivante :

1. Récupère le processeur virtuel courant dans **vp**
2. Récupère le thread courant dans **current**, et le prochain thread devant s'exécuter sur **vp** dans **next**
3. Si **next** est NULL (<=> pas d'autre thread prêt pour ce processeur virtuel), alors on recherche un thread prêt sur les autres processeurs virtuels. Si un tel thread existe, il est retiré de la file d'attente de l'autre processeur virtuel et est stocké dans **next** (ceci permet l'équilibrage des charges entre les processeurs virtuels)
4. Si un thread a été marqué comme devant être ré-ordonnancé (attribut **resched** de **vp** non NULL), alors il est ajouté dans la liste des threads prêt à l'exécution du processeur virtuel vp.
5. Si **current** n'est pas **idle**,
   (a) s'il est dans l'état **RUNNING**, alors il ré-ordonnancé dans la file d'exécution du processeur virtuel
   (b) si **next** est toujours NULL (aucun thread prêt sur aucuns des processeurs virtuels), alors **idle** est assigné à **next**
6. Si **next** est non NULL, et que **current** est différent de **next**, alors l'ordonnanceur effectue un changement de context : **next** prends la main sur le processeur virtuel. Après l'appel de *mthread_mctx_swap()*, c'est un nouveau thread qui a la main sur le processeur virtuel.
7. Finalement, avant de retourner, l'ordonannceur récupère le processeur virtuel courant (après le changement de context, donc). Si un *spinlock* a été enregistré comme devant être dévérouillé, il est dévérouillé ici (attribut **p** de **vp**).

2.1.2 Listes d'ordonnancement (2.3 et 2.4)

Les fonctions concernant la gestion des listes se situent dans le fichier mthread.c, et sont :
— *mthread_list_init()* : initialise une liste
— *mthread_insert_first()* : ajoute un élément en tête de liste
— *mthread_insert_last()* : ajoute un élément en fin de liste
— *mthread_remove_first()* : supprime l'élément en tête de liste
Pour la clareté du code, j'ai également ajouté une fonction sur les listes
— *mthread_is_empty()* : renvoie vrai si la liste est vide, faux sinon

2.1.3 Bloquer un thread (2.5)

Pour bloquer un thread, il suffit de passer son état dans **BLOCKED**, puis d'appeler l'ordonnanceur pour que le thread laisse la main.

Pseudo code simplifié :

```
struct mthread_s * thread = ...; //ex: mthread_self()
thread->status = BLOCKED;
mthread_yield();
```

Chapitre 3

Les mutex

3.1 Code d'exemple (3.1)

FIGURE 3.1 – Schéma simplifié d'un processus multithread

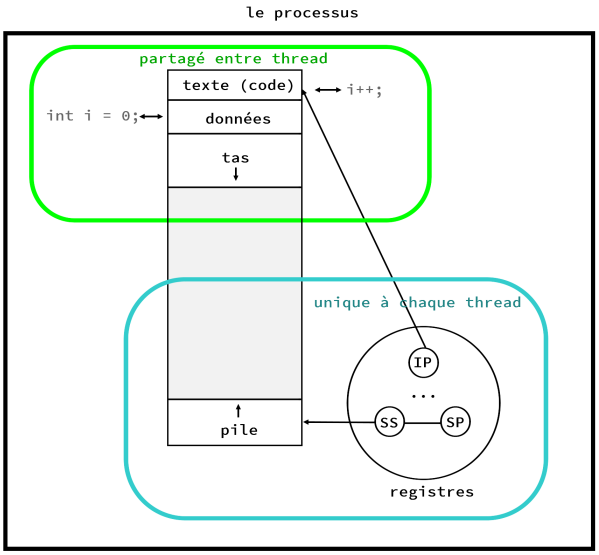
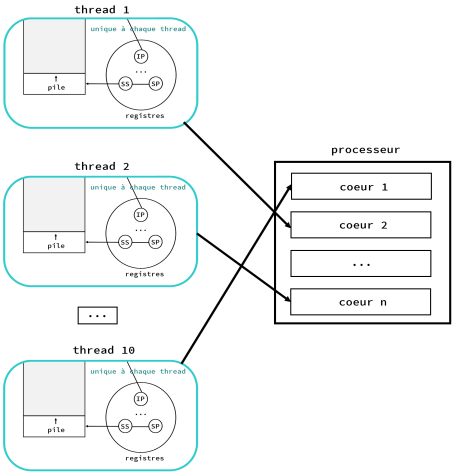


FIGURE 3.2 – Attachement des threads à un processeur multicoeur



Les threads d'un même processus partagent **le segment data** du programme. Ici, la variable i est situé dans le segment data.

Les threads ne partagent pas la pile d'execution et les registres : 2 threads peuvent donc être dans la même portion de code.

Dans notre cas, si 2 threads se situent sur l'instruction $i++;$, alors le comportement sera indéterminé, car 2 threads accèdent en parallèle à la même zone mémoire.

Dans notre exemple bien précis, l'utilisation d'un mutex permet d'assurer qu'il n'y aura jamais plus d'un thread dans la portion de code entre le verrou ($i++;$).

3.2 Implémentation des mutex

Les mutex étaient pré-implémentés, mais leur implémentation a été revu. Le code a été commenté des modifications effectuées

3.2.1 mthread_mutex_init (3.2)

Pour pouvoir implémenter l'initialiseur statique par la suite, cette fonction met simplement à 0 les 3 attributs de la structure du mutex.

3.2.2 mthread_mutex_lock (3.3)

Modification apporté : l'attribut **mutex->list** est alloué en mémoire lorsque le 1er thread se bloque.

3.2.3 mthread_mutex_unlock (3.4)

Modification apporté : l'attribut **mutex->list** est dé-alloué de la mémoire lorsque le dernier thread bloqué se débloque.

3.2.4 mthread_mutex_destroy (3.5)

Aucunes modifications apportés.

3.3 Implémentation bonus des mutex

3.3.1 mthread_mutex_trylock (3.7)

L'implémentation suit la description de la norme POSIX.

3.3.2 PTHREAD_MUTEX_INITIALIZER (3.8)

Initialise statiquement un mutex, trivial :

```
# define MTHREAD_MUTEX_INITIALIZER  {0, 0, NULL}
```

Utilisation (alloué à la compilation dans le segment DATA) :

```
pthread_mutex_t mutex = MTHREAD_MUTEX_INITIALIZER;

int main() {
    return 0;
}
```

Utilisation (alloué sur la pile) :

```
int main() {
    pthread_mutex_t mutex = MTHREAD_MUTEX_INITIALIZER;
    [...]
    return 0;
}
```

3.4 Demonstration (3.6)

Les tests concernant les mutex sont préfixés de **mutex_** (ex : **mutex_init**)

Chapitre 4

Les sémaphores

4.1 Code d'exemple (2.1)

Dans cet exemple, le sémaphore est utilisé comme un mutex (sa valeur vaut initialement 1).
2 threads sont crées, et chaque threads affichent 20 fois, son nom 10 fois suivi d'un retour à la ligne.
Le sémaphore permet ici d'assurer que chaque ligne ne contiennent le nom que d'un seul thread.
Exemple d'affichage possible :

```
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
```

ou encore :

```
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
```

ou encore :

```
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
BBB BBB BBB BBB BBB BBB BBB BBB BBB BBB
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
AAA AAA AAA AAA AAA AAA AAA AAA AAA AAA
```

4.2 Implémentation des sémaphores

4.2.1 mthread_sem_init (2.2)

Tous les attributs sont initialisés à 0.

4.2.2 mthread_sem_wait (2.3)

Implémentation similaire à *mthread_mutex_lock*.

4.2.3 mthread_sem_post (2.4)

Implémentation similaire à *mthread_mutex_unlock*.

4.2.4 mthread_sem_destroy (2.5)

Implémentation similaire à *mthread_mutex_lock*.

4.3 Implémentation bonus des sémaphores

4.3.1 mthread_sem_trylock (2.7)

Implémentation similaire à *mthread_try_lock*.

4.3.2 mthread_sem_getvalue (2.8)

Implémentation trivial :

```
1 int mthread_sem_getvalue(mthread_sem_t * sem, unsigned int * sval) {
2     *sval = sem->value;
3     return 0;
4 }
```

4.4 Demonstration (2.6)

Les tests concernant les semaphores sont préfixés de **sem_** (ex : **sem_init**)

Chapitre 5

Les conditions et les clés

5.1 Implémentation des conditions

Le code a été amplement commenté (voir **mthread_cond.c**)

5.1.1 *mthread_cond_init (2.1)*

5.1.2 *mthread_cond_wait (2.2)*

5.1.3 *mthread_cond_signal (2.3)*

5.1.4 *mthread_cond_broadcast (2.4)*

5.1.5 *mthread_cond_destroy (2.5)*

5.2 Demonstration (2.6)

Les tests sont situés dans le dossier **tests** à la racine du projet.
Pour compiler les tests, il suffit de taper la commande :

```
1 > make
```

Les tests concernant lesconditions sont préfixés de **sem_** (ex : **sem_init**)

5.3 Implémentation des clés posix

Afin d'implémenter correctement les clés de thread POSIX, la structure d'un thread a été légèrement modifié, j'y ai ajouté 3 attributs :

```
struct mthread_s {
    [...]

    /* ajout pour gerer les cles */
    /* tableau de valeurs */
    struct mthread_s_key {
        void * value;
        void (*destr_f)(void *);
    } * keys;
    /* taille du tableau */
    unsigned int nb_keys;
    /* index de la prochaine [U+FFFD]ibre dans le tableau */
    unsigned int next_key;
};
```

La structure est géré de la façon suivante en mémoire :
— les clés sont stockées dans un tableau de structure *struct mthread_s_key*
— l'indice de la clé dans le tableau correspond à sa valeur
— l'attribut *next_key* correspond à la 1ère clé libre dans le tableau
— lorsqu'une clé est libéré, elle est ajouté à la liste des clés libres
— la liste des clés libres et la liste des clés utilisées partagent le même espace mémoire. L'implémentation optimise le nombre d'allocation mémoire (détail ci dessous).

Cette structure de donnée permet d'effectuer toutes les opérations sur les clés en **O(1)**
(avec la complexité **n** le nombre de clés définies)

5.3.1 Gestion de la mémoire

On considère le cas d'utilisation suivante :

1. Création d'une clé
2. Création d'une clé
3. Création d'une clé
4. Destruction de la clé 2.
5. Création d'une clé

Voici l'état de la mémoire dans les états successifs :
TODO : Schéma
La valeur des clés crées est dans la suivante selon les états :

1. Valeur de la clé créée : **0**
2. Valeur de la clé créée : **1**
3. Valeur de la clé créée : **2**
4. Libération de la clé **1**
5. Valeur de la clé créée : **1**

Le code a été amplementé commenté, et l'implémentation suit la logique mémoire expliqué ci dessus.

5.3.2 *mthread_key_create (2.7)*

5.3.3 *mthread_key_delete (2.8)*

5.3.4 *mthread_key_setspecific (2.9)*

5.3.5 *mthread_key_getspecific (2.10)*

5.4 Demonstration (2.11)

Les tests concernant les clés sont préfixés de **key_** (ex : **key_create**)