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

5 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/pthread/mthread

Table des matières

Chapitre 1

Découverte de la bibliothèque mthread

1.1 L'ordonnanceur

1.1.1 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**).

1.1.2 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

1.1.3 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 2

Les mutex

2.1 Code d'exemple (3.1)

2.2 Implémentation des mutex

2.2.1 mthread_mutex_init (3.2)

2.2.2 mthread_mutex_lock (3.3)

2.2.3 mthread_mutex_unlock (3.4)

2.2.4 mthread_mutex_destroy (3.5)

2.3 Demonstration (3.6)

2.4 Implémentation bonus des mutex

2.4.1 mthread_mutex_trylock (3.7)

2.4.2 PTHREAD_MUTEX_INITIALIZER (3.8)

Chapitre 3

Les sémaphores

3.1 Code d'exemple (2.1)

3.2 Implémentation des sémaphores

3.2.1 mthread_sem_init (2.2)

3.2.2 mthread_sem_wait (2.3)

3.2.3 mthread_sem_post (2.4)

3.2.4 mthread_sem_destroy (2.5)

3.3 Demonstration (2.6)

3.4 Implémentation bonus des sémaphores

3.4.1 mthread_sem_trylock (2.7)

3.4.2 mthread_sem_getvalue (2.8)

Chapitre 4

Les conditions et les clés

4.1 Implémentation des conditions

4.1.1 mthread_cond_init (2.1)

4.1.2 mthread_cond_wait (2.2)

4.1.3 mthread_cond_signal (2.3)

4.1.4 mthread_cond_broadcast (2.4)

4.1.5 mthread_cond_destroy (2.5)

4.2 Demonstration (2.6)

4.3 Implémentation des clés posix

4.3.1 mthread_key_create (2.7)

4.3.2 mthread_key_delete (2.8)

4.3.3 mthread_key_setspecific (2.9)

4.3.4 mthread_key_getspecific (2.10)

4.4 Demonstration (2.11)

Chapitre 5

Bilan