TP 9

Processus légers (threads) sémaphores binaires (mutex)

Threads POSIX.

Un processus UNIX est constitué d'un ensemble d'informations permettant au système d'assurer le contrôle des ressources du processus (descripteurs des fichiers ouverts, implantation en mémoire, etc.) ou son exécution (ordonnanceur, pile, signaux).

Contrairement à la fonction `fork()` qui effectue une recopie d'un processus en mémoire, le mécanisme des threads (on parle aussi d'activités) permet de ne dupliquer que les informations relatives à l'exécution du processus. Les informations relatives aux ressources restent communes à toutes les activités d'un processus. Lors de la création de threads par un processus, les threads possèderont le même *pid*, mais des identités d'activités (*tid*) différentes.

– Option de compilation : `-lpthread`
– Include : `#include <pthread.h>`

Fonctions POSIX de manipulation des processus légers (threads).

Création

```
int pthread_create(pthread_t * tid,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

Crée un nouveau thread et lui fait exécuter la fonction `start_routine` en lui passant `arg` comme premier argument. L'identifiant du thread est enregistré dans `tid`. Si `attr` est `NULL`, les attributs par défaut sont utilisés (thread joignable, ordonnancement standard).

Self

```
pthread_t pthread_self(void)
```

Renvoie l'id du thread courant.

Attente de la mort d'un autre thread

```
int pthread_join(pthread_t tid,
                 void **thread_return);
```

Suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par `tid` achève son exécution, soit en appelant `pthread_exit()` soit après avoir été annulé. Si `thread_return` ne vaut pas NULL, la valeur renvoyée par `tid` y sera enregistrée.

Détacher

```
int pthread_detach(pthread_t tid);
```

Place le thread `tid` dans l'état détaché (par opposition à joignable). Cela garantit que les ressources mémoires consommées par `tid` (pile, etc.) seront immédiatement libérées lorsque l'exécution de `tid` s'achèvera. Cependant, cela empêche les autres threads de se synchroniser sur la mort de `tid` en utilisant `pthread_join()`.

Terminaison

```
void pthread_exit(void *retval);
```

Termine l'exécution du thread appelant. L'argument `retval` est la valeur de retour du thread. Si le thread est joignable, cette valeur de retour peut être consultée par un autre thread en utisant `pthread_join()`. Appelée dans le thread initial cette fonction a pour effet d'attendre la fin de l'exécution des autres threads. Une appel implicite à `pthread_exit()` est fait lorsqu'un thread autre que le thread initial sort de la fonction `start_routine` qui avait été utilisée pour le créer.

Annulation

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *etat_pred);
int pthread_setcanceltype(int mode, int *ancien_mode);
```

L'annulation est le mécanisme par lequel un thread peut interrompre l'exécution d'un autre thread. C'est l'équivalent pour les threads de la fonction `kill()` des processus classiques. Plus précisément, un thread peut envoyer une requête d'annulation à un autre thread. Selon sa configuration (état et type d'annulation), le thread visé peut soit ignorer la requête (état `PTHREAD_CANCEL_DISABLE`), soit l'honorer immédiatement (type `PTHREAD_CANCEL_ASYNCHRONOUS`), soit enfin retarder son application jusqu'à ce qu'un point d'annulation soit atteint (c'est le comportement par défaut).

Manipulation 1.
Écrire un programme qui crée cinq threads (en plus du thread initial) et fait exécuter à chacun une fonction `tache(void * i)` *pour i entier allant de 1 à 5. Dnas cette fonction, faire une boucle vide de 10000 étapes, puis l'affichage à l'écran du numéro i et de l'id du thread. Le thread initial réalisera également l'affichage. Vérifier sur cet exemple que :*

1. *L'utilisation de la fonction* `exit()` *dans un des threads du processus entraîne la terminaison de l'ensemble des threads.*
2. *La terminaison du thread initial entraîne la libération de toutes les ressources, donc la terminaison de l'ensemble des threads du processus. Comment éviter cette terminaison prématurée ?*

Manipulation 2 (Thread et fork)**.**
Que se passe t-il lorsqu'un processus ayant plusieurs threads fait un `fork()` *dans l'un de ses threads, les threads sont-ils dupliqués ? Tester (faire un thread qui fait une lecture bloquante sur l'entrée standard et un thread qui fait le* `fork()`*).*

Exclusion mutuelle par sémaphore binaire.

Un mutex est un objet d'exclusion mutuelle (MUTual EXclusion), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des sections critiques.

Un mutex peut ête dans deux états : déverrouillé (pris par aucun thread) ou verrouillé (appartenant à un thread). Un mutex ne peut être pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

Initialisation

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Initialise le mutex pointé par `mutex` selon les attributs de mutex spécifié par `mutexattr`. Si `mutexattr` vaut `NULL`, les paramètres par défaut sont utilisés (mutex de type rapide).

On peut également initialiser un mutex rapide de manière statique :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Il y a deux autres types de mutex disponibles (voir la page de man).

Verrouillage

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Déverrouillage

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Destruction d'un mutex non verrouillé

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Exemple. Une variable globale partagée x peut être protégée par un mutex comme suit :

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Tous les accès et modifications de `x` doivent être entourés de paires d'appels à `pthread_mutex_lock` et `pthread_mutex_unlock` comme suit :

```
pthread_mutex_lock(&mut);
/* agir sur x */
pthread_mutex_unlock(&mut);
```

Manipulation 3.
Cinq philosophes dînent et discutent autour d'une table ronde. Chacun a devant lui un plat de nouilles chinoises et deux baguettes de chaque coté de l'assiette. Malheureusement entre deux assiettes il n'y a qu'une baguette : pour chaque philosophe, sa baguette de gauche est partagée avec son voisin de gauche et sa baguette de droite est partagée avec son voisin de droite.

Pour manger un philosophe a besoin des deux baguettes. Chaque philosophe veut alternativement manger et discuter, ceci indéfiniement. On représentera un philosophe par un thread.

Comment éviter un interblocage en utilisant un seul mutex ? Indication : pour qu'un philosophe puisse manger il suffit que sont voisin de gauche et son voisin de droite ne soient pas déjà en train de manger. On ne cherche pas à éviter les situations de famine (un philosophe qui n'arrive jamais à manger).