Module Système L3/S6 – TD/TP no 4
Activités (threads)

Vous pouvez récupérer du code ici :
`http ://www.lisyc.univ-brest.fr/pages_perso/rodin/FTP/Enseignements/L3/Systeme/`
Le fichier à récupérer est : `TD-TP4.tar.gz`
Pour décompresser l'archive faire : `tar zxvf TD-TP4.tar.gz` ou `gunzip TD-TP4.tar.gz` et `tar xvf TD-TP4.tar`

1 Rappels sur les threads

Sous Unix, le programmeur dispose d'une bibliothèque de gestion des activités : la bibliothèque normalisée Posix. Cette bibliothèque se nomme `libpthread.a`. Il faut donc compiler vos programmes avec `-lpthread`. Certaines fonctions concernant l'ordonnanceur (comme `sched_yield`) sont disponibles dans une autre bibliothèque : `librt.a`. Dans ce cas, il faut indiquer en plus `-lrt`.

1.1 Partie 1 : précautions

Les threads sont des activités concurrentes accédant à un espace d'adressage commun. Il est donc nécessaire de les synchroniser avec des sémaphores de type `pthread_mutex_t` (voir plus loin).

Toujours à cause de l'espace d'adressage commun, un autre problème se pose. Il faut veiller à écrire des **fonctions réentrantes**, c'est à dire des fonctions sans donnée statique. Rappel : une donnée statique dans une fonction existe en 1 seul exemplaire durant toute la durée du programme.

- **Exemple de fonction non réentrante**
  - Contenu de `buffer` indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
static char buffer[0x10];
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```


- **Version réentrante de cette fonction**
  - Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

- **Choix de primitives systèmes réentrantes**
  - Fonction ayant l'extention `_r` (`Posix.1c`)
  - Ex : `asctime()` et `asctime_r()`

1.2 Partie 2 : création, terminaison, attente

1.2.1 création

La fonction `pthread_create` crée et initialise une nouvelle activité.

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
                   void * (*routine)(void *), void *arg);
```

En entrée, la fonction reçoit :
- l'adresse d'un `pthread_t`, `pthread_t` qui contiendra l'identifiant du thread créé,
- l'adresse d'un `pthread_attr_t`, `pthread_attr_t` qui contient des attributs de création du thread ; pour un usage "courant", on peut indiquer `NULL`,
- l'adresse de la fonction contenant le programme exécuté dans l'activité,
- un argument "utilisateur" `arg` indiqué sous la forme d'un pointeur sur une zone mémoire ; ce pointeur sera transmis à la fonction exécutée par le thread.

Cette fonction `pthread_create` retourne un code éventuel d'erreur de création.

1.2.2 terminaison

La fonction `pthread_exit` provoque l'arrêt de l'activité qui l'exécute.

```
#include <pthread.h>
void pthread_exit(void * value_ptr);
```

En entrée, cette fonction reçoit une adresse qui sera transmise (on peut indiquer NULL) à une autre activité attendant la fin de l'activité courante (voir `pthread_join`).

Remarque : la fin normale (avec `return`) de la fonction exécutée par l'activité est équivalente à un `pthread_exit`.

1.2.3 attente

La fonction `pthread_join`, fonction de synchronisation, permet de bloquer une activité en attente de la fin d'une autre activité.

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status);
```

En entrée, la fonction reçoit :
- l'identifiant du thread à attendre,
- l'adresse d'un pointeur de pointeur... permettant de récupérer l'adresse indiquée lors du `pthread_exit` (ou du `return`).

1.3 Partie 3 : synchronisation

Les threads étant des activités concurrentes, il faut les synchroniser vis à vis de l'accès à des variables partagée... il faut des sémaphores ! $\Longrightarrow$ `pthread_mutex_t`

Il peut également être intéressant d'avoir un mécanisme permettant à un thread d'avertir d'autres threads, qu'il a effecté un changement de la valeur d'une variable partagée... $\Longrightarrow$ `pthread_cond_t`.

Signalons qu'une variable condition de type `pthread_cond_t` est une variable partagée par les différents threads... Il faut donc un `pthread_mutex_t` associé !

1.3.1 Les sémaphores d'exclusion mutuelle : `pthread_mutex_t`

- **Création d'un verrou** (`man 3 pthread_mutex_init`)
  - Type : `pthread_mutex_t`
  - Initialisation statique
    - `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER ;`
  - Initialisation dynamique
    - `int pthread_mutex_init(pthread_mutex_t * mtx, pthread_mutexattr_t * attr) ;`
  - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'un verrou** (`man 3 pthread_mutex_destroy`)
  - `int pthread_mutex_destroy(pthread_mutex_t * mtx) ;`
  - Le verrou n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si le verrou est monopolisé → erreur `EBUSY`

- **Opération de verrouillage** (`man 3 pthread_mutex_lock`)
  - `int pthread_mutex_lock(pthread_mutex_t * mtx) ;`
  - Si le verrou est libre
    - il est monopolisé par le thread courant
    - le thread courant poursuit son traitement
  - Si le verrou n'est pas libre
    - le thread courant est bloqué jusqu'à la libération du verrou
  - Retour : `0` si ok, non nul si erreur
- **Opération de déverrouillage** (`man 3 pthread_mutex_unlock`)
  - `int pthread_mutex_unlock(pthread_mutex_t * mtx) ;`
  - Libère le verrou
  - Si des threads sont bloqués en attente sur ce verrou
    - l'un d'eux est débloqué et monopolise le verrou
  - Retour : `0` si ok, non nul si erreur
- **Tentative de verrouillage** (`man 3 pthread_mutex_trylock`)
  - `int pthread_mutex_trylock(pthread_mutex_t * mtx) ;`
  - Si le verrou est libre
    - il est monopolisé par cet appel qui retourne `0`
  - Si le verrou n'est pas libre
    - cet appel retourne immédiatement un résultat non nul
    - il ne faut pas utiliser les données protégées par le verrou

1.3.2 Les variables conditions : `pthread_cond_t`

- **Création d'une condition** (`man 3 pthread_cond_init`)
  - Type : `pthread_cond_t` (doit être associé à un `mutex`)
  - Initialisation statique
    - `pthread_cond_t myCond=PTHREAD_COND_INITIALIZER ;`
  - Initialisation dynamique
    - `int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t * attr) ;`
  - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'une condition** (`man 3 pthread_cond_destroy`)
  - `int pthread_cond_destroy(pthread_cond_t * cond) ;`
  - La condition n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si la condition est utilisée → erreur `EBUSY`

- **Attente d'une condition** (`man 3 pthread_cond_wait`)
  - `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mtx) ;`
  - Bloque le thread courant
    → débloqué quand une modification est signalée sur `cond`
    → évite l'attente active
  - `cond` n'a aucune valeur logique ! (sert à la synchronisation)
  - `mtx` doit être monopolisé avant et libéré après
  - Interruptible par les signaux → relance
  - **Démarche usuelle :**

```
pthread_mutex_lock(&mtx) ;
while( !conditionIsSatisfied())
  pthread_cond_wait(&cond,&mtx) ;
pthread_mutex_unlock(&mtx) ;
```

- **Attente temporisée d'une condition** (`man 3 pthread_cond_timedwait`)
  - `int pthread_cond_timedwait(pthread_cond_t * cond, pthread_mutex_t * mtx, const struct timespec * date) ;`
  - Même principe que `pthread_cond_wait()`
  - Renvoie `ETIMEDOUT` si `date` est dépassée
  - `date` est une limite, pas un délai !
    - Prendre la date courante (`time()`, `gettimeofday()`)
    - Ajouter un délai (structure `timespec` de `nanosleep()`)
    - Basé sur le temps universel (pas de fuseau horaire)
    - Voir le cours sur la mesure du temps

- **Signaler une condition** (`man 3 pthread_cond_broadcast`)
  - `int pthread_cond_broadcast(pthread_cond_t * cond) ;`
  - Débloque tous les threads attendant la condition `cond`
  - Le verrou associé à `cond` doit être monopolisé avant et libéré après
  - **Démarche usuelle :**

```
pthread_mutex_lock(&mtx) ;
/* make this condition become true */
pthread_cond_broadcast(&cond) ;
pthread_mutex_unlock(&mtx) ;
```

  - `int pthread_cond_signal(pthread_cond_t * cond) ;`
  - Ne débloque qu'un thread parmi ceux qui attendent `cond`
  - Un peu plus efficace que `pthread_cond_broadcast`
  - Bien moins général (incohérence si plusieurs threads en attente !)

2 Exercice 1 (TD) : création, terminaison, attente

1. Écrire un programme `calculfactorielleLocal.c` qui crée deux activités, pour calculer indépendamment la factorielle de 2 nombres.
   Chaque activité exécute le code décrit dans la fonction `calculerFactorielle`. Cette fonction reçoit en paramètre l'adresse d'un entier pour lequel la factorielle doit être calculée.
   Le calcul est effectué dans une variable locale de l'activité.
   Le résultat est communiqué au thread principal via l'adresse d'une zone allouée dynamiquement (via un `malloc`) dans `calculerFactorielle`. Cette zone doit bien sûr être libérée (via un `free`) dans le thread principal une fois que le résultat est affiché par celui-ci.
2. Modifier le programme précédent pour que les calculs soient effectués dans **UNE variable globale** ; les activités se terminent alors sans argument de sortie. Ecrire pour cela un programme `calculfactorielleGlobal.c`. Alors, ça marche ?

3 Exercice 2 (TD-TP) : synchronisation

Nous souhaitons ici créer des threads, synchroniser leurs exécutions et attendre leurs fins.

3.1 Produit scalaire multi-threads

Nous souhaitons réaliser le produit scalaire (somme de produits) de deux vecteurs à l'aide d'une multitude de threads. Si les vecteurs sont de dimension `N` il y aura `N` threads dédiés aux multiplications et un thread dédié à la somme.

Lorsque les deux vecteurs sont initialisés par l'activité principale, les threads de multiplication peuvent effectuer leurs calculs. Lorsque ces multiplications ont **toutes** été effectuées, le thread d'addition peut commencer son calcul. Quand cette somme est calculée l'activité principale est alors en mesure d'afficher le résultat.

Nous souhaitons de plus que notre programme soit capable de réaliser plusieurs produits scalaires les uns après les autres (sur des vecteurs de même taille). Les threads de multiplication et d'addition seront donc créés une fois pour toutes au début du programme et tourneront en boucle pour chaque nouveau produit scalaire à calculer.

La synchronisation des différentes activités (threads + `main()`) aura lieu autour d'une unique structure `Product` contenant les données suivantes :

- `state` : l'étape au sein du produit scalaire courant
- `pendingMult` : les multiplications non terminées
- `cond` : la variable condition synchronisant les évolutions des champs précédents
- `mutex` : le verrou associé à la contition précédente
- `nbIteration` : le nombre d'iterations à effectuer
- `size` : la taille des vecteurs
- `v1` et `v2` : les vecteurs dont il faut calculer le produit scalaire
- `v3` : le vecteur contenant les résultats des multiplications
- `result` : le résultat du produit scalaire

L'état du produit scalaire courant est représenté par les constantes suivantes :
- `STATE_WAIT` : début du programme
- `STATE_MULT` : les multiplications peuvent commencer
- `STATE_ADD` : l'addition peut commencer
- `STATE_PRINT` : l'affichage du résultat peut avoir lieu

Une instance de la structure `Product` est communiquée aux différents threads via une variable globale. Pour le thread d'addition, cette information est suffisante, mais pour les threads de multiplication, il est nécessaire d'indiquer à quel index des vecteurs le thread concerné doit calculer.

Il faudra passer au thread concerné l'adresse d'un entier contenant l'index... Il doit exister un entier différent pour chaque thread de multiplication.

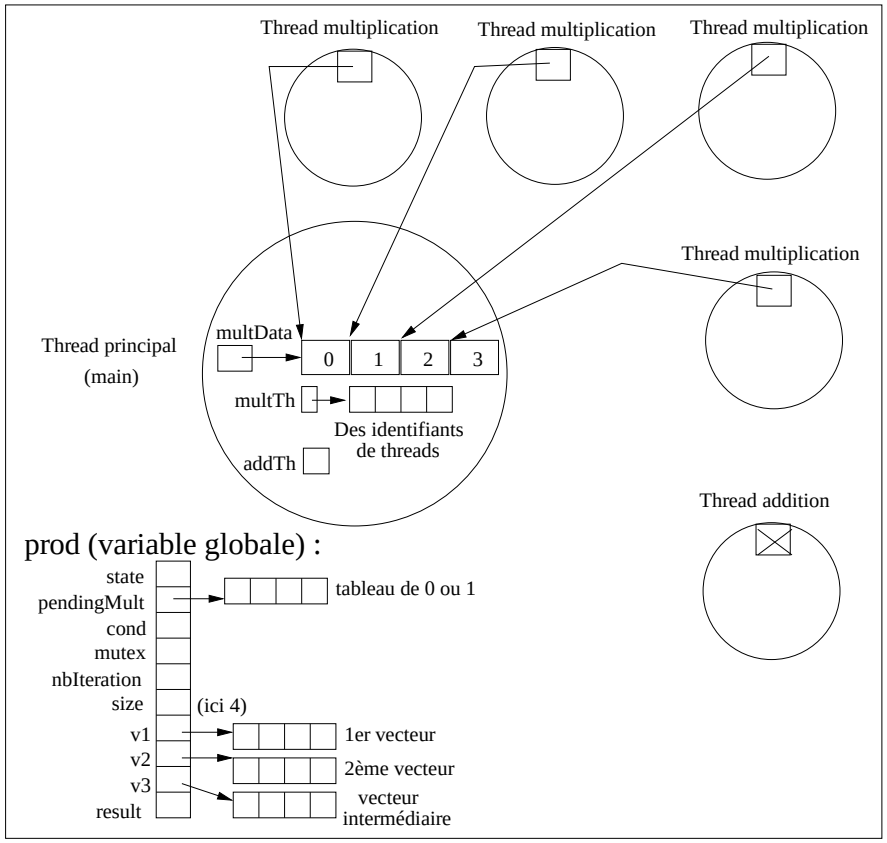

FIG. 1 – Principales structures de données utilisées

Les opérations de multiplication et d'addition ayant une durée tellement courte, nous pourrons éventuellement simuler de longs calculs en utilisant la fonction `wasteTime()`. Celle-ci réalise une attente active afin de ne pas perturber l'ordonnanceur par la mise en sommeil des threads.

Les algorithmes des traitements à effectuer sont donnés dans le code source à compléter. Il faudra veiller à bien attendre que toutes les multiplications soient terminées (fonction `nbPendingMult()`) avant de commencer l'opération d'addition. C'est le rôle du champ `pendingMult` qui doit être initialisé (fonction `initPendingMult()`) avant les multiplications et qui doit être modifié à chaque fois qu'une multiplication se termine. Les manipulations des champs `state` et `pendingMult` devront **TOUJOURS** avoir lieu sous le contrôle des champs `cond` et `mutex` sans quoi certaines modifications risquent de ne pas être détectées.

Exemple d'exécution :

```
$ ./produit
usage: ./produit nbIteration vectorSize
$ ./produit 3 4
Begin mult(0)
Begin mult(1)
Begin mult(2)
Begin mult(3)
Begin add()
--> mult(0)
--> mult(2)
--> mult(1)
--> mult(3)
<-- mult(2) : 0.632*1.51=0.956
<-- mult(0) : 0.751*2.16=1.63
<-- mult(1) : -4.14*-3.5=14.5
<-- mult(3) : -1.76*3.52=-6.19
--> add
<-- add
ITERATION 0, RESULT=10.9
--> mult(1)
--> mult(0)
--> mult(2)
--> mult(3)
<-- mult(2) : -3.47*-3.84=13.3
<-- mult(1) : 0.678*-3.11=-2.11
<-- mult(3) : -1.59*1.72=-2.75
<-- mult(0) : -0.863*2.48=-2.14
--> add
<-- add
ITERATION 1, RESULT=6.32
--> mult(3)
--> mult(1)
--> mult(2)
--> mult(0)
<-- mult(2) : 3.76*0.553=2.08
Quit mult(2)
<-- mult(0) : -3.64*-3.89=14.1
Quit mult(0)
<-- mult(3) : -0.181*4.62=-0.838
Quit mult(3)
<-- mult(1) : 4.8*2.65=12.7
--> add
Quit mult(1)
<-- add
ITERATION 2, RESULT=28.1
Quit add()
$
```

3.2 Squelette du programme

Le squelette du programme est disponible dans l'archive `TD-TP4.tar.gz`.

```
#include <pthread.h>          /* produit.c */
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
```