TP3&4 M2LSE SEE
La norme POSIX

La bibliothèque *pthread* correspond à une implémentation de la norme POSIX1003.1c. Elle fournit des primitives pour créer et gérer des threads / processus légers. Dans le domaine des threads du monde unixien, cette norme s'est imposée.
Une documentation plus complète de la librairie *pthreads* est disponible dans le man (`man libpthread`, `man pthread_xxx` et/ou `man pthreads`).

Premiers pas …

Voici un premier programme utilisant certaines fonctions POSIX :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NB_THREADS 3

void *travailUtile(void *null)
{
   int i;
   double resultat=0.0;
   for (i=0; i<1000000; i++)
   {
     resultat = resultat + (double)random();
   }
   printf("resultat = %e\n",resultat);
   pthread_exit((void *) 0);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NB_THREADS];
   pthread_attr_t attr;
   int rc, t;
   void *status;

   /* Initialisation et activation d'attributs */
   pthread_attr_init(&attr); //valeur par défaut
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); //attente du thread possible

   for(t=0; t<NB_THREADS; t++)
   {
      printf("Creation du thread %d\n", t);
      rc = pthread_create(&thread[t], &attr, travailUtile, NULL);
      if (rc)
      {
         printf("ERROR; le code de retour de pthread_create() est %d\n", rc);
         exit(-1);
      }
   }

   /* liberation des attributs et attente de la terminaison des threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NB_THREADS; t++)
   {
      rc = pthread_join(thread[t], &status);
      if (rc)
      {
         printf("ERROR; le code de retour du pthread_join() est %d\n", rc);
         exit(-1);
```

```
        }
        printf("le join a fini avec le thread %d et a donne le status= %ld\n",t, (long)status);
    }

    pthread_exit(NULL);
}
```

Ce programme crée 3 threads qui font des choses très utiles et attend leur terminaison.

1. Testez ce programme (la compilation de programme utilisant des threads POSIX se fait avec l'option –lpthread).
2. Modifiez ce programme pour afficher la valeur par défaut de la taille de la pile (stack) des threads. Augmentez-la ensuite d'un méga octet pour les threads crées (avant leur création).
3. Modifiez le programme pour passer en argument à chaque thread la valeur qu'il doit additionner à la variable *resultat*, cette valeur doit être différente pour chaque thread.

Besoin de synchronisation

1. Créez un programme qui utilise 4 threads pour le calcul d'un produit scalaire (2 vecteurs de 400 éléments chacun par exemple). Chaque thread fera un quart du produit qu'il stockera dans une variable temporaire. La modification du résultat global du produit scalaire par chaque thread sera effectuée dans une section critique protégée par un *mutex* (sémaphore binaire). Ce résultat sera affiché à la fin de l'exécution des 4 threads.
2. Nous souhaitons à présent faire afficher le résultat par un thread différent. Pour cela chaque thread incrémente un compteur. Le dernier thread (le nombre total est connu) relâche une variable conditionnelle qui permet au thread d'affichage de faire son boulot.
3. Nous voulons à présent utiliser une mémoire partagée pour stocker les 2 vecteurs, les résultats intermédiaires, le résultat final ainsi que le compteur. Pour faciliter la tâche, nous créerons une structure qui contient l'ensemble de ces éléments. La compilation doit s'effectuer avec l'option –lrt (inclusion de la librairie temps réel). Une mémoire partagée :
    - est créée avec la fonction `shm_open()`,
    - sa taille est augmentée avec `ftruncate()`,
    - elle est intégrée dans l'espace d'adressage d'une tâche avec `mmap()`,
    - chaque processus (non pas thread) qui fini d'utiliser la mémoire ferme le descripteur (retourné par `shm_open`) avec la fonction `close()`,
    - chaque processus peut tenter de détruire la mémoire partagée, si toutefois aucun processus n'y accède, avec la fonction `shm_unlink()`,
    - le processus peut aussi enlever l'espace mémoire de son espace d'adressage avec la fonction `munmap()`.

Files de messages

Refaire l'exercice précédent de la section « Besoin de synchronisation » en utilisant les files de messages.

Entrées/Sorties asynchrones et signaux

Les E/S asynchrones présentent l'avantage de pouvoir initier une requête d'E/S (par exemple une lecture d'un fichier sur disque) et de faire autre chose en attendant la complétion de cette dernière.

La structure de base qu'utilisent les API aio (asynchronous Input/Output) est la structure ***aiocb.*** Elle caractérise l'entrée /sortie à effectuer et contient un certain nombre de champs dont (man aio):

- `aio_nbytes` : le nombre d'octets à lire ou écrire par l'E/S asynchrone,
- `aio_filedes` : le fichier sur lequel l'E/S doit être effectuée,
- `aio_buf` : le tampon devant contenir les données lues ou les données à écrire,
- `aio_offset` : le déplacement à partir du début du fichier.

Voici un exemple montrant une utilisation basique d'une E/S asynchrone :

```
#include <aio.h>
#include <stdio.h>


int main(int argc, char * argv[])
{
          if(argc != 2)
          {
                    printf("Usage: %s {filename}\n", argv[0]);
                    return -1;
          }
          struct aiocb cb; // bloc de contrôle de l'E/S asynchrone
          struct aiocb * cbs[1];
          //Ouverture du fichier (spécifié en argument) sur lequel l'E/S va être effectuée
          FILE * file = fopen(argv[1], "r+");

          //definition du bloc de contrôle de l'entrée/sortie
          cb.aio_buf = malloc(11);
          cb.aio_fildes = fileno(file); //récupérer le descripteur d'un fichier à partir de  son nom
          cb.aio_nbytes = 10;
          cb.aio_offset = 0;

          //lancer la lecture
          aio_read(&cb);

          //Suspension du processus dans l'attente de la terminaison de la lecture
          cbs[0] = &cb;
          aio_suspend(cbs, 1, NULL);

          printf("operation AIO a retouree %d\n", aio_return(&cb));

          return 0;
}
```

Il est évident que ce programme ne fait rien d'autre qu'implémenter une lecture synchrone avec les API des E/S asynchrones. L'idéal serait de lancer la lecture et de continuer à faire autre chose et être notifié de la fin de l'opération d'E/S par l'OS.

Pour qu'un signal soit envoyé au processus effectuant une E/S asynchrone lorsque celle-ci est terminée, on utilise le champ (une structure) `aio_sigevent` de la structure `aiocb`, cette structure définie la notification de la complétion de l'E/S demandée. Elle est constituée des champs suivants :

- `sigev_notify` : le mécanisme utilisé pour la notification. Pour les programmes mono thread la valeur doit être à SIGEV_SIGNAL (sinon SIGEV_THREAD).

- `sigev_signo` : le numéro de signal à envoyer. Il existe un ensemble de signaux (temps réels) utilisables par l'application qui se situent entre les macros SIGTERMIN et SIGTERMAX qui se trouvent définies dans signal.h.
- `sigev_value` : union d'un entier et d'un pointeur délivré au *handler* du message.

Pour utiliser ce type de mécanisme, il faut définir dans la structure de contrôle de l'E/S le signal (entre SIGRTMIN et SIGRTMAX) qui va nous notifier la complétion de cette dernière et définir une fonction à exécuter à la réception de ce signal avec la primitive `sigaction()`. Dans `sigev_value`, on peut mettre une information à envoyer avec le signal pour, par exemple, différencier les E/S dont on reçoit le signal de complétion.

Faites un programme (mono thread) qui lance 2 lectures et une écriture sur des fichiers (différents ou pas) et qui affiche ce qui a été lu pour les lectures et un acquittement pour l'écriture.

"*Flash back*"

Reprenez l'exercice du produit scalaire, mais le scénario sera, cette fois-ci, plus réaliste. En effet, on suppose que les vecteurs de données se trouvent dans des fichiers différents.

1. Créez 2 fichiers où chacun contient l'un les vecteurs dont on calcule le produit scalaire.
2. Créez un programme qui envoie 4 requêtes d'E/S asynchrones pour lire les 4 moitiés (ou quarts) de vecteur à donner pour traitement à 2 threads que l'on créera dès réception des données. Un dernier thread affichera le résultat. Choisissez le moyen de communication qui vous convienne pour l'envoi des données aux threads.

Annexe

Groupe fonctionnel	Quelques fonctions principales
Multi-Threads	pthread_create, pthread_exit, pthread_join, pthread_detach, pthread_equal, pthread_self, pthread_once
Ordonnancement de processus et de Threads	pthread_getschedparam, pthread_setschedparam, pthread_setschedprio, pthread_yield,
Signaux temps réel	sigqueue, pthread_kill, sigaction, sigaltstack, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember, sigwait, sigwitinfo, sigtimedwait pthread_sigmask
Synchro et comm inter processus	mq_open, mq_close, mq_receive, mq_send, mq_unlink, mq_timedsend, mq_timedreceive, mq_notify, mq_getattr, mq_setattr, sem_init, sem_destroy, sem_open, sem_close, sem_unlink, sem_wait, sem_trywait, sem_timedwait, sem_post, sem_getvalue, pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_setprioceiling pthread_mutex_getprioceiling pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_mutex_timedlock, pthread_cond_init, pthread_cond_destroy, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_signal, pthread_cond_broadcast,

	shm_open, shm_close, shm_unlink, mmap, munmap
Données spécifiques aux threads	pthread_key_create, pthread_key_delete, pthread_get_specific, pthread_set_specific.
Différents attributs	pthread_attr_init pthread_attr_destroy pthread_attr_setdetachstate pthread_attr_getdetachstate pthread_attr_getstackaddr pthread_attr_getstacksize pthread_attr_setstackaddr pthread_attr_setstacksize pthread_attr_getschedparam pthread_attr_setschedparam pthread_attr_getschedpolicy pthread_attr_setschedpolicy pthread_attr_setinheritsched pthread_attr_getinheritsched pthread_attr_setscope pthread_attr_getscope pthread_mutexattr_init pthread_mutexattr_destroy pthread_mutexattr_init pthread_mutexattr_destroy pthread_mutexattr_getpshared pthread_mutexattr_setpshared pthread_mutexattr_getprotocol pthread_mutexattr_setprotocol pthread_mutexattr_setprioceiling pthread_mutexattr_getprioceiling pthread_condattr_init pthread_condattr_destroy pthread_condattr_getpshared pthread_condattr_setpshared
Gestion Mémoire	mlock, mlockall, munlock, munlockall, mprotect
E/S Async	aio_read, aio_write, lio_listio, aio_error, aio_return, aio_fsync, aio_suspend, aio_cancel.
Horloges et minuteries (*timers*)	clock_gettime, clock_settime, clock_getres, timer_create, timer_delete, timer_getoverrun, timer_gettime, timer_settime.
Annulation	pthread_cancel, pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel, pthread_cleanup_push, pthread_cleanup_pop,