1 Définition

Les processus nécessitent souvent des traitements parallèles. Un des moyens pour cela consiste à créer un ou plusieurs processus fils avec fork(). Cette façon de faire a des inconvénients :

- le temps de création d'un processus fils est assez long ;
- les données sont séparées et ne peuvent être partagées ;
- la communication interprocessus est quand même assez complexe.

Une autre solution consiste à créer ce qu'on appelle un (ou des) processus léger : un (des) thread(s). L'avantage c'est que l'exécution du thread correspond en gros à l'exécution d'une fonction qui se ferait en parallèle avec le programme du main(). Les données globales seront partagées entre le programme principal appelé aussi thread principal et les autres threads.

2 La programmation des threads

2.1 Compilation d'un programme multithreads

Toutes les fonctions relatives aux threads sont définies dans le fichier d'en-tête <pthread.h> et le code machine se trouve dans la bibliothèque libpthread.a (soit -lpthread à la compilation) :

```
gcc -lpthread -o prog main.c
```

2.2 Le lancement d'un thread

Pour créer un thread, il faut déclarer une variable le représentant. Celle-ci sera de type pthread_t

Ensuite if faut une fonction à exécuter dans le thread. Cette dernière devra être de la forme :

void *fonction(void* arg)

et contiendra le code à exécuter par le thread.

Enfin il faut lancer le thread avec la primitive pthread_create() dont voici le prototype :

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

Paramètres :

- thread : l'adresse de la variable qui recevra l'identité du thread lancé ;
- attr : l'adresse d'une structure permettant de passer des attributs. On mettra NULL et le thread sera créé avec ses attributs par défaut.
- start_routine : la fonction qui contient le code du thread.
- arg : adresse du bloc de données passé en paramètres à la fonction.

Valeur retournée : En cas de succès, l'identifiant du nouveau thread est stocké à l'emplacement mémoire pointé par l'argument thread, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

2.3 La terminaison d'un thread

La fonction qui s'exécute dans le thread doit faire appel à la primitive pthread_exit() (et pas simplement faire appel à exit). Voici le prototype :

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

Paramètre :

- retval : adresse du premier octet d'une zone mémoire contenant les données à retourner(voir la primitive pthread_join plus bas).

valeur retournée : aucune, la fonction pthread_exit() ne rend jamais la main

2.4 Synchroniser le processus avec la fin d'un thread

Comme pour le processus père qui attend que le fils se termine, un thread (par exemple le thread principal) peut avoir besoin d'attendre qu'un thread se termine. Pour cela il y a la primitive pthread_join(). Son prototype :

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

Paramètres :
- th : le thread dont on veut attendre la fin
- thread_return : Si thread_return ne vaut pas NULL, la valeur renvoyée par th y sera enregistrée. Cette valeur sera l'argument passé à pthread_exit

Valeur retournée : En cas de succès, le code renvoyé par th est enregistré à l'emplacement pointé par thread_return, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

2.5 Exemple

Voici déjà le code du thread : c'est une fonction baptisée thread_1. Dans cet exemple le programme principal passera comme argument un entier, que le thread (pour l'exemple) saisit :

```
void *thread_1(void *arg)
 {
 int* pval=(int*)arg;
 printf("Nous sommes dans le thread, tapez un entier:");
 scanf("%d",pval);
 printf("val=%d\n",*pval);
 pthread_exit(NULL);
 }
```

On ne retournera aucune valeur par l'appel de pthread_exit.

Voici le main() :

```
int main(void)
{
int valeur;
pthread_t thread1;
printf("Avant la création du thread.\n");
if (pthread_create(&thread1, NULL, thread_1, &valeur))
  {
        printf("pthread_create\n");
        exit(0);
  }

if (pthread_join(thread1,NULL))
  {
        printf("pthread_join\n");
        exit(0);
  }
printf("Après la fin du thread. Valeur=%d \n",valeur);
return 0;
}
```

A la fin le main() affichera la valeur saisie.

3 Les mutex.

3.1 Problématique.

Avec les threads, toutes les variables sont partagées : c'est la mémoire partagée. Mais cela pose des problèmes. En effet, quand deux threads cherchent à modifier deux variables en même temps, que se passe-t-il ? Et si un thread lit une variable quand un autre thread la modifie ? Il faut un mécanisme de synchronisation : les **mutex**, un des outils permettant l'exclusion mutuelle.

Un mutex est une variable de type **pthread_mutex_t**. Elle va nous servir de verrou, pour nous permettre de protéger des données. Ce verrou peut donc prendre deux états : disponible et verrouillé. Quand un thread a accès à une variable protégée par un mutex, on dit qu'il **tient le mutex**. Bien évidemment, il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

3.2 Accès au mutex.

Lorsque plusieurs threads se partagent des variables se pose le problème de l'accès à ces variables, également du point de vue de leur visibilité (portée). Une solution consiste à utiliser des variables globales. On pourrait donc mettre aussi le mutex qui protègera ces variables en global. On utilisera plutôt une structure contenant d'une part la variable (ou les variables) à partager et le mutex lui-même. On a vu dans la section précédente que la création du thread s'accompagne d'un argument pointeur arg que l'on fera pointer sur la structure en question.

Par exemple :

```
typedef struct data {
    int valeur;
    pthread_mutex_t mutex;
} data;
```

3.3 Initialiser un mutex

Conventionnellement, on initialise un mutex avec la primitive **pthread_mutex_init()**, déclarée dans **pthread.h**.

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

Paramètres :
- mutex : le mutex à initialiser ;
- mutexattr : pointeur de type pthread_attr_t, permettant de fixer des attributs particulier au mutex. En pratique on mettra NULL et le mutex sera créé avec les attributs par défaut.

Valeur de retour : pthread_mutex_init() retourne toujours 0.

Voici un exemple de code concernant la déclaration et l'initialisation du mutex :

```
int main(void)
{
    data ma_data; //variable à partager avec le thread
   pthread_mutex_init( &(ma_data.mutex),NULL);
```

3.4 Verrouiller et déverrouiller un mutex

Si dans un thread on veut manipuler la variable partagée et s'assurer qu'aucun autre thread n'accèdera à la variable on définit une section critique dans le code du thread. C'est donc une portion de code où le thread doit être le seul à accéder à la variable. On va verrouiller le mutex eu début de cette section critique et le déverrouiller à la fin de celle-ci.

Les deux primitives permettant de verrouiller et déverrouiller le mutex sont :

```
int pthread_mutex_lock(pthread_mutex_t *mut);
int pthread_mutex_unlock(pthread_mutex_t *mut);
```

On passe en paramètre les adresses des mutex.

L'opération pthread_mutex_lock bloque tant que le mutex n'est pas déverrouillé.