Les threads

Voici un chapitre à ne manquer sous aucun prétexte car c'est une des bases de la programmation système sous Unix : l'étude des *threadsPosix* (*Posix* est le nom d'une famille de standards qui indique un code standard).

Qu'est ce qu'un thread ?

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un **thread** de contrôle unique, le **thread principal**. Du point de vue programmation, ce dernier exécute le `main`.

Vous avez pu remarquer, lors de notre étude des processus, qu'en général, le système réserve un processus à chaque application, sauf quelques exceptions. Beaucoup de programmes exécutent plusieurs activités en parallèle, du moins en apparent parallélisme, comme nous l'avons vu précédemment. Comme à l'échelle des processus, certaines de ces activités peuvent se bloquer, et ainsi réserver ce blocage à un seul thread séquentiel, permettant par conséquent de ne pas stopper toute l'application.
Ensuite, il faut savoir que le principal avantage des threads par rapport aux processus, c'est la facilité et la rapidité de leur création. En effet, tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Cela évite donc l'allocation de tous ces espaces lors de la création, et il est à noter que, sur de nombreux systèmes, la création d'un thread est environ cent fois plus rapide que celle d'un processus.
Au-delà de la création, la superposition de l'exécution des activités dans une même application permet une importante accélération quant au fonctionnement de cette dernière.

Sachez également que la communication entre les threads est plus aisée que celle entre les processus, pour lesquels on doit utiliser des notions compliquées comme les tubes (voir chapitre suivant).

Le mot « *thread* » est un terme anglais qui peut se traduire par « *fil*

d'exécution ». L'appellation de « *processus léger* » est également utilisée.

Compilation

Toutes les fonctions relatives aux *threads* sont incluses dans le fichier d'en-tête `<pthread.h>` et dans la bibliothèque `libpthread.a` (soit `-lpthread` à la compilation).

Exemple :

Écrivez la ligne de commande qui vous permet de compiler votre programme sur les *threads* constitué d'un seul fichier `main.c` et avoir en sortie un exécutable nommé `monProgramme`.

Correction :

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

Et n'oubliez pas d'ajouter `#include <pthread.h>` au début de vos fichiers.

Manipulation des threads

Créer un thread

Pour créer un thread, il faut déjà déclarer une variable le représentant. Celle-ci sera de type `pthread_t` (qui est, sur la plupart des systèmes, un `typedef` d'`unsigned long int`).
Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

```
#include <pthread.h>


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

Ce prototype est un peu compliqué, c'est pourquoi nous allons récapituler ensemble.

- La fonction renvoie une valeur de type `int` : 0 si la création a été réussie ou une autre valeur si il y a eu une erreur.
- Le premier argument est un pointeur vers l'identifiant du thread (valeur de type `pthread_t`).
- Le second argument désigne les attributs du thread. Vous pouvez choisir de mettre le thread en état joignable (par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel…). Dans nos exemple, on mettra généralement NULL.
- Le troisième argument est un pointeur vers la 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, le quatrième et dernier argument est l'argument à passer au thread.

Supprimer un thread

Et qui dit créer dit supprimer à la fin de l'utilisation.
Cette fois, ce n'est pas une fonction casse-tête :

```
#include <pthread.h>


void pthread_exit(void *ret);
```

Elle prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

Première application

Voici un premier code qui réutilise toutes les notions des threads que nous avons vu jusque là.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
    perror("pthread_create");
    return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");
```

```
    return EXIT_SUCCESS;
}
```

On compile, on exécute. Et là, zut... Le résultat, dans le meilleur des cas, affiche le message de thread en dernier. Dans le pire des cas, celui-ci ne s'affiche même pas (ce qui veut dire que le `return` s'est exécuté avant le thread...).
Ce qui normal, puisqu'en théorie, comme avec les processus, le thread principal ne va pas attendre de lui-même que le thread se termine avant d'exécuter le reste de son code.
Par conséquent, il va falloir lui en faire la demande.  Pour cela, Dieu pthread a créé la fonction `pthread_join`.

Attendre la fin d'un thread

Voici le prototype de cette fameuse fonction :

```
#include <pthread.h>

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

Elle prend donc en paramètre l'identifiant du thread et son second paramètre, un pointeur, permet de récupérer la valeur retournée par la fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de `pthread_exit`).

Exercice résumé

Écrivez un programme qui crée un thread demandant un nombre à l'utilisateur, l'incrémentant une fois puis l'affichant dans le main.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if (pthread_create(&thread1, NULL, thread_1, NULL)) {
    perror("pthread_create");
    return EXIT_FAILURE;
    }

    if (pthread_join(thread1, NULL)) {
    perror("pthread_join");
    return EXIT_FAILURE;
```

```
    }

    printf("Après la création du thread.\n");

    return EXIT_SUCCESS;
}
```

Et le résultat est enfin uniforme :

```
Avant la création du thread.
Nous sommes dans le thread.
Après la création du thread.
```

Exclusions mutuelles

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 ?
C'est assez problématique. Par conséquent, nous allons voir une mécanisme de synchronisation : les **mutex**, un des outils permettant l'**exclusion mutuelle**.

Les mutex

Concrètement, un **mutex** est en C 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.

Le problème, c'est qu'il faut que le mutex soit accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale. Mais nous ne sommes pas des barbares ! Par conséquent, j'ai choisi de vous montrer une autre solution : déclarer le mutex dans une structure avec la donnée à protéger. Allez, un petit exemple ne vous fera pas de mal :

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

Ainsi, nous pourrons passer la structure en paramètre à nos threads, grâce à la fonction `pthread_create`.

Passons à la pratique : comment manipuler les mutex grâce à pthread ?

Initialiser un mutex

Conventionnellement, on initialise un mutex avec la valeur de la constante PTHREAD_MUTEX_INITIALIZER, déclarée dans `pthread.h`.

```
#include <stdlib.h>
#include <pthread.h>

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

```
int main(void)

{

    data new_data;


    new_data.mutex = PTHREAD_MUTEX_INITIALIZER;


    return EXIT_SUCCESS;

}
```

Verrouiller un mutex

L'étape suivante consiste à établir une **zone critique**, c'est-à-dire la zone où plusieurs threads ont l'occasion de modifier ou de lire une même variable en même temps.
Une fois cela fait, on verrouille le mutex grâce à la fonction :

```
#include <pthread.h>


int pthread_mutex_lock(pthread_mutex_t *mut);
```

Déverrouiller un mutex

A la fin de la zone critique, il suffit de déverrouiller le mutex.

```
#include <pthread.h>


int pthread_mutex_unlock(pthread_mutex_t *mut);
```

Détruire un mutex

Une fois le travail du mutex terminé, on peut le détruire :

```
#include <pthread.h>


int pthread_mutex_destroy(pthread_mutex_t *mut);
```

Les conditions

Lorsqu'un *thread* doit patienter jusqu'à ce qu'un événement survienne dans un autre *thread*, on emploie une technique appelée la condition.

Quand un *thread* est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre *thread*.

Comme avec les *mutex*, on déclare la condition en variable globale, de cette manière :

```
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un *mutex* :

```
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t *nomMut
```

Pour réveiller un *thread* en attente d'une condition, on utilise la fonction :

```
int pthread_cond_signal(pthread_cond_t *nomCondition);
```

Exemple :

Créez un code qui crée deux *threads* : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

Correction :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de la cond
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du mutex */

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
    pthread_t monThreadCompteur;
    pthread_t monThreadAlarme;

    pthread_create (&monThreadCompteur, NULL, threadCompteur, (void*)NULL);
    pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL); /*

    pthread_join (monThreadCompteur, NULL);
    pthread_join (monThreadAlarme, NULL); /* Attente de la fin des threads

    return 0;
}
```

```
void* threadCompteur (void* arg)
{
    int compteur = 0, nombre = 0;

    srand(time(NULL));

    while(1) /* Boucle infinie */
    {
        nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
        compteur += nombre; /* On ajoute ce nombre à la variable compteur *

        printf("\n%d", compteur);

        if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
        {
            pthread_mutex_lock (&mutex); /* On verrouille le mutex */
            pthread_cond_signal (&condition); /* On délivre le signal : con
            pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

            compteur = 0; /* On remet la variable compteur à 0 */
        }

        sleep (1); /* On laisse 1 seconde de repos */
    }

    pthread_exit(NULL); /* Fin du thread */
}

void* threadAlarme (void* arg)
```

```
{
    while(1) /* Boucle infinie */
    {
        pthread_mutex_lock(&mutex); /* On verrouille le mutex */
        pthread_cond_wait (&condition, &mutex); /* On attend que la conditi
        printf("\nLE COMPTEUR A DÉPASSÉ 20.");
        pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
    }

    pthread_exit(NULL); /* Fin du thread */
}
```

Résultat :

```
lucas@lucas-Deskstop:~/Documents$ ./monprog

4
9
18
26
LE COMPTEUR A DÉPASSÉ 20.
9
18
23
LE COMPTEUR A DÉPASSÉ 20.
0
3
5
9
12
19
23
LE COMPTEUR A DÉPASSÉ 20.
0
8
9
10
17
```

```
25
LE COMPTEUR A DÉPASSÉ 20.
2
10
10
16
25
LE COMPTEUR A DÉPASSÉ 20.
8
10
18
26
LE COMPTEUR A DÉPASSÉ 20.
0
7
9
^C
```

Terminons ce chapitre par quelques moyen mnémotechniques qui peuvent vous permettre de retenir toutes les notions que l'on a apprises :

- Vous pouvez remarquer que toutes les variables et les fonctions sur les threads commencent par **pthread_**
- pthread_create : *create* = *créer* en anglais (donc *créer le thread)*
- pthread_exit : *exit* = *sortir* (donc *sortir du thread*)
- pthread_join : *join* = *joindre* (donc *joindre le thread*)
- PTHREAD_MUTEX_INITIALIZER : *initializer* = *initialiser* (donc *initialiser le mutex*)
- pthread_mutex_lock : *lock* = *vérouiller* (donc *vérouiller le mutex*)
- pthread_mutex_unlock : *unlock* = *dévérouiller* (donc *dévérouiller le mutex*)
- pthread_cond_wait : *wait* = *attendre* (donc *attendre la condition*)

(Comme quoi l'anglais ça sert des fois )