

Voici une illustration de la gestion des threads POSIX, nous allons construire une fonction comme suite :

```
void* lire_entier(void *arg) {
    int un_entier;
    int val_arg = (int) arg;
    printf("Bienvenue chez le thread ayant comme argument %d\n", val_arg);
    printf("Priere de saisir un entier:");
    scanf("%d", &un_entier);
    pthread_exit((void *) un_entier);
}
```

Cette fonction reçoit comme paramètre numéro de thread, ce thread permet d'afficher un message de bienvenue et de lire un entier au clavier, puis de le retourner au thread principal qui est main :

```
int main(void) {
    int i, val_retour;
    pthread_t thread1;
    srand(time(NULL));
    i = 1 + rand() % 100;
    int succes = pthread_create(&thread1, NULL, lire_entier, (void *) i);
    if (succes != 0) {
        fprintf(stderr, "Erreur de creation du thread ...");
        exit(0);
    }
    pthread_join(thread1, (void *)&val_retour);
    printf("Vous avez lu une valeur entiere egale a %d\n", val_retour);
    return 0;
}
```

Le thread principal initialise un nouveau thread nommé « thread1 », La fonction « srand » permet d'initialiser le générateur de nombres aléatoires (la fonction rand) fournit par la librairie C standard. Après avoir générer un nombre aléatoirement, nous allons créer notre deuxième thread avec la fonction suivante :

```
int pthread_create(
    pthread_t *thread_id,
    pthread_attr_t *thread_attr,
    void *(*thread_func)(void *),
    void *arg
);
```

Qui prend les paramètres suivants : l'adresse du thread créé, spécifier que les attributs par défaut peuvent être utilisés par placer NULL, la fonction au-dessus qui va lire un entier et finalement l'argument passé à la fonction « lire\_entier » lors de l'appel.

Si la création est effectuée correctement, nous allons attendre la fin de l'exécution de notre thread par la fonction « pthread\_join » :

```
int pthread_join(
    pthread_t thread_id,
    void **value_returned
);
```

Cette fonction qui prend dans sa première paramètre l'identifiant du thread nous permet de stocker « value\_returned » la valeur retournée par la fonction « pthread\_exit » du thread à attendre dans une variable qu'on a créé dans le thread principal.

```
void pthread_exit(
    void *value_returned,
);
```

Maintenant, on peut afficher la valeur retournée par le thread.

```
Bienvenue chez le thread ayant comme argument 18
Priere de saisir un entier:50
Vous avez lu une valeur entiere egale a 50

-----
Process exited after 10.73 seconds with return value 0
Appuyez sur une touche pour continuer...
```

On peut ajouter d'autres threads pour afficher même valeur en hexadécimal et octal :

```
pthread_join(thread1, (void *)&val_retour);
printf("Vous avez lu une valeur entiere egale a : %d\n", val_retour);
pthread_t thread2,thread3;
pthread_create(&thread2, NULL, afficher_octal, (void *) val_retour);
pthread_create(&thread3, NULL, afficher_hex, (void *) val_retour);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
```

```
void* afficher_octal(void *arg) {
    printf("Vous avez lu une valeur octal egale a : %o\n", (int) arg);
}

void* afficher_hex(void *arg) {
    printf("Vous avez lu une valeur hexadecimal egale a : %x\n", (int) arg);
}
```

Nous allons modifier le programme pour qu'il affiche le temps pris par chaque opération, la lecture, l'affichage et aussi son temps de réponse depuis son lancement jusqu'à terminaison.

```
Bienvenue chez le thread ayant comme argument 9
Priere de saisir un entier:12
Temps pour la lecture : 4028
Vous avez lu une valeur entiere egale a : 12
Vous avez lu une valeur octal egale a : 14
Vous avez lu une valeur hexadecimal egale a : c
Temps pour l'affichage' : 1

-----
Process exited after 4.093 seconds with return value 0
Appuyez sur une touche pour continuer...
```

## Synchronisation des pthreads

Il existe principalement trois mécanismes pour assurer la synchronisation entre les pthreads, à savoir : les mutexes, les sémaphores, et les variables de conditions.

### a) Les mutexes

Les threads POSIX fonctionnent de manière asynchrone, donc des mutex (ou verrous en français) peuvent être utilisés pour la synchronisation. Étant donné que les threads partagent des variables globales et des descripteurs de fichiers ouverts les uns avec les autres, un verrou mutex doit être introduit pour la gestion des accès simultanés afin de garantir la cohérence des données causée par un contrôle incorrect des parties clés. Les verrous Mutex sont des verrous fournis par la bibliothèque Pthreads.

On va déclarer un mutex, et l'initialisation du mutex se fait avant la création des threads de la manière suivante :

```
pthread_mutex_t mutex;

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Le premier paramètre sera rempli à l'initialisation et sera utilisé comme identifiant pour ce mutex. Le deuxième paramètre permet de choisir un certain nombre d'attributs pour le mutex initialisé. Lorsqu'un thread souhaite accéder à une section critique, il verrouille le mutex. Si un autre thread se trouve dans la section critique, le thread sera bloqué jusqu'à la libération du mutex.

### b) Les sémaphores

Les sémaphores sont un type « sem\_t » et un ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive.

```
#include <semaphore.h>
sem_t sem;
int sem_init (sem_t *sem, int shared, unsigned int value);
int sem_destroy (sem_t *sem);
int sem_wait (sem_t *sem);
int sem_post (sem_t *sem);
int sem_getvalue (sem_t *sem, int *value);
int sem_trywait(sem_t *sem);
```

Pour utiliser un sémaphore, on doit tout d'abord inclure la bibliothèque « semaphore », on peut créer et initialiser le sémaphore avec la fonction qui prend comme paramètres l'adresse du sémaphore, indication si le sémaphore peut être partagé ou non et la valeur initiale de ce sémaphore. Après utilisation, il faut systématiquement libérer le sémaphore en appelant la fonction destroy. Les autres fonctions permettant respectivement décrémenter le compteur et passe à l'instruction suivante, incrémenter le compteur, récupérer la valeur du compteur dans une variable passée par adresse et le thread appelant teste si le sémaphore est non nul et décrémente le sémaphore mais sans se bloquer si ce dernier est nul.

### c) Les variables de conditions

Les variables de condition de type « pthread\_cond\_t ». De telles variables permettent de mettre en attente un ou plusieurs threads jusqu'à ce qu'un événement donné se produise.

```
pthread_cond_t condition;
int pthread_cond_init(pthread_cond_t *condition, pthread_condattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *condition);
int pthread_cond_broadcast(pthread_cond_t *condition);
int pthread_cond_destroy(pthread_cond_t *condition);
```

L'initialisation de la condition se fait avec la fonction « init » qui prend comme paramètres l'adresse de la condition créée et attributs pour la condition initialisée. Lorsqu'un thread souhaite attendre une condition, il effectue cette série d'appel (mutex\_lock -> cond\_wait -> mutex\_unlock) ceci nous permet de protéger la fonction « cond\_wait » et lorsqu'un thread constate la réalisation de l'événement, une notification est alors envoyée à un ou à tous les threads en attente de l'événement.

Si on veut réveiller un seul thread bloqué sur une variable de condition on peut utiliser la fonction « cond\_signal », si par contre, on veut réveiller tous les threads en attente, on doit plutôt utiliser la fonction « cond\_broadcast ». Une variable de condition qui n'est plus utilisée doit être libérée par « cond\_destroy ».

### EXEMPLE 2

Dans l'exemple suivant, nous allons exécuter un programme qui permet d'illustrer la synchronisation entre les threads en utilisant des mutexes et des variables de conditions. Le but du programme est de gérer des transactions bancaires sur un compte à condition le solde du compte ne doit jamais descendre au-dessous de zéro.

```

Creation du thread de la banque!
Creation des threads clients !
Client 1 prend 0 dirhams, reste 200 en solde!
Client 2 prend 0 dirhams, reste 200 en solde!
Client 0 prend 0 dirhams, reste 200 en solde!
Client 4 prend 0 dirhams, reste 200 en solde!
Client 3 prend 0 dirhams, reste 200 en solde!
Client 1 prend 11 dirhams, reste 189 en solde!
Client 2 prend 11 dirhams, reste 178 en solde!
Client 0 prend 11 dirhams, reste 167 en solde!
Client 3 prend 11 dirhams, reste 156 en solde!
Client 4 prend 11 dirhams, reste 145 en solde!
Client 1 prend 35 dirhams, reste 110 en solde!
Client 2 prend 35 dirhams, reste 75 en solde!
Client 0 prend 35 dirhams, reste 40 en solde!
Client 4 prend 35 dirhams, reste 5 en solde!
      Alimentation du compte bancaire avec 200 dirhams!
Client 3 prend 35 dirhams, reste 165 en solde!
Client 1 prend 21 dirhams, reste 144 en solde!
Client 2 prend 21 dirhams, reste 123 en solde!
Client 0 prend 21 dirhams, reste 102 en solde!
Client 4 prend 21 dirhams, reste 81 en solde!
Client 3 prend 21 dirhams, reste 60 en solde!

```

Comme vous voyez le solde initial du compte bancaire est 200dh. Ensuite, chaque client prend une somme d'argent qui est générée aléatoirement entre 1 et 50 dirhams. Si la valeur du solde bancaire est insuffisante pour débiter la valeur générée, on le crédite d'un montant de 200 dirhams.

Nous allons utiliser cinq threads pour simuler les dépenses et un thread pour la banque. On commence par la création de thread pour la banque qui va être bloqué par une variable de condition, après les cinq threads pour clients seront créés.

```

pthread_mutex_lock (& bank.mutex_balance);
if (val > bank.balance) {
    pthread_cond_signal (& bank.cond_balance);
    pthread_cond_wait (& bank.cond_clients, & bank.mutex_balance);
} /* fin if */
bank.balance = bank.balance - val;
printf("Client %d prend %d dirhams, reste %d en solde!\n", nb, val, bank.balance);
pthread_mutex_unlock (& bank.mutex_balance);

```

Chaque thread demande la permission d'entrer dans la zone critique par verrouiller « mutex\_balance » pour qu'il peut accéder à la Bank, si la valeur à débiter est disponible, l'opération s'effectue correctement et il laisse le verrou à un autre thread. Si non, il faut réveiller thread de Bank qui est bloqué sur une variable de condition « cond\_balance » :

```

pthread_mutex_lock (& bank.mutex_balance);
pthread_cond_wait (& bank.cond_balance, & bank.mutex_balance);
bank.balance = INITIAL_BALANCE;
printf ("\t\t Alimentation du compte bancaire avec %d dirhams!\n", bank.balance);
pthread_cond_signal (& bank.cond_clients);
pthread_mutex_unlock (& bank.mutex_balance);

```

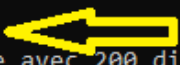
Ce thread lui-même sera bloqué sur la variable de condition « cond\_clients » qui attend thread Bank finir son alimentation du solde. Après faire alimenter le solde, thread de Bank fait une notification pour qu'il permet à ce thread client de continuer son opération qui sert à prendre la somme d'argent qui veut.

Si on n'utilise pas les variables de condition, la probabilité pour que le verrou « mutex\_balance » soit occupé par thread client est très petit, alors le solde va être presque toujours égal à 200dhs.

```
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Client 0 prend 10 dirhams, reste 190 en solde!
Client 3 prend 10 dirhams, reste 180 en solde!
Client 4 prend 10 dirhams, reste 170 en solde!
Client 2 prend 10 dirhams, reste 160 en solde!
Client 1 prend 10 dirhams, reste 150 en solde!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
```

Lorsque enlève les mutexes, le solde de compte bancaire après chaque transaction peut être erroné ; parce que plus qu'un deux threads concurrents essaient en même temps de modifier une variable globale qui est dans notre cas le solde, un thread modifie le solde bancaire tandis qu'un autre thread essaie de la lire.

```
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
Client 4 prend 35 dirhams, reste 130 en solde!
Client 1 prend 35 dirhams, reste 95 en solde!
Client 2 prend 35 dirhams, reste 60 en solde!
Client 3 prend 35 dirhams, reste 25 en solde!
Client 0 prend 35 dirhams, reste 165 en solde!
Alimentation du compte bancaire avec 200 dirhams!
Alimentation du compte bancaire avec 200 dirhams!
```



Maintenant, nous allons modifier notre programme pour qu'on peut mettre une limite à notre bancaire pour fixer le plafond de ces dépenses à 1000 dirhams. Après on exécute notre programme :

```
Client 0 prend 30 dirhams, reste 155 en solde!
Client 1 prend 4 dirhams, reste 151 en solde!
Client 1 prend 7 dirhams, reste 144 en solde!
impossible de prend 2 dirhams, la limite de 1000dh par jour!
impossible de prend 1 dirhams, la limite de 1000dh par jour!
Client 1 prend 5 dirhams, reste 139 en solde!
Client 0 prend 8 dirhams, reste 131 en solde!
Client 1 prend 0 dirhams, reste 131 en solde!
impossible de prend 2 dirhams, la limite de 1000dh par jour!
impossible de prend 0 dirhams, la limite de 1000dh par jour!
impossible de prend 1 dirhams, la limite de 1000dh par jour!
Client 2 prend 8 dirhams, reste 123 en solde!
impossible de prend 0 dirhams, la limite de 1000dh par jour!
impossible de prend 2 dirhams, la limite de 1000dh par jour!
impossible de prend 1 dirhams, la limite de 1000dh par jour!
impossible de prend 2 dirhams, la limite de 1000dh par jour!
```

On peut bien voir le temps pris par chaque transaction, on observe que si le solde est insuffisant la transaction prend plus de temps qu'une simple transaction.

```
Client 0 prend 29 dirhams, reste 126 en solde!
                                temps prise par transaction : 997
Client 1 prend 29 dirhams, reste 97 en solde!
                                temps prise par transaction : 1
Client 3 prend 29 dirhams, reste 68 en solde!
                                temps prise par transaction : 1
Client 2 prend 29 dirhams, reste 39 en solde!
                                temps prise par transaction : 0
Client 4 prend 29 dirhams, reste 10 en solde!
                                temps prise par transaction : 1
                                Alimentation du compte bancaire avec 200 dirhams!
Client 0 prend 17 dirhams, reste 183 en solde!
                                temps prise par transaction : 2998
Client 3 prend 17 dirhams, reste 166 en solde!
                                temps prise par transaction : 1
Client 2 prend 17 dirhams, reste 149 en solde!
                                temps prise par transaction : 1
```

## Communication entre les pthreads

Dans ce qui suit, nous allons donner un aperçu général sur les principaux mécanismes qu'on peut utiliser pour la communication entre les threads :

- ✚ La première façon pour communiquer avec un thread qu'on a déjà vu dans la première partie dans ce TP est d'utiliser les arguments de la fonction de démarrage du thread lors de sa création par la fonction « **pthread\_create** » et aussi la valeur retournée par le thread via l'appel à « **pthread\_join** ».
- ✚ On peut faire la communication entre les threads par une mémoire partagée. En effet, tous les threads d'un processus ont accès aux mêmes variables globales et au même heap, alors n'importe quel thread peut modifier et lire les variables globales partagées.
- ✚ Le troisième mécanisme de communication entre threads est de passer par les signaux déjà définis par le système d'exploitation. Les signaux sont des primitifs systèmes identifiés par un numéro de 0 à 63, et à chaque signal une application peut y associer une fonction, dite communément handler, qui sera appelée lorsque le signal sera reçu par un thread (l'appel système signal).
- ✚ Un mécanisme de communication qui n'est pas recommandé c'est tout simplement l'utilisation des fichiers qui peut même fonctionner si les processus sont exécutés consécutivement et non simultanément. Mais il possède quand même plusieurs inconvénients.
- ✚ Les pipes peuvent être l'un des moyens de communication entre processus/threads. Une pipe se compose de deux descripteurs de fichiers connectés entre eux : tout ce qui est écrit sur le premier sera lu sur le deuxième.
- ✚ Le sixième moyen de communication entre processus/threads est les sockets UNIX qui peuvent être considérés similaires aux tubes/pipes nommés. Les sockets UNIX sont prévus dans une architecture client-serveur, avec un serveur unique qui contrôle le socket, et de nombreux clients qui peuvent communiquer avec lui.
- ✚ Le dernier moyen de communication entre processus/threads est la file de messages, le mécanisme MQ fonctionne comme un service de gestion de files. Les files contiennent des messages avec des priorités variables, et les messages les plus prioritaires et les plus anciens sont lus en premier. Les cinq principales fonctions de base pour la manipulation de POSIX MQ sont : « **mq\_open** », « **mq\_receive** », « **mq\_send** », « **mq\_close** », et « **mq\_unlink** ».

### EXEMPLE 3 :

Code ne fonctionne pas

Etude de cas