# TP n°3 : Threads et sémaphores

## 1 Utilisation des threads Linux

Les threads de LINUX sont une implantation normalisée des P-threads (threads Posix). Ces threads sont gérés à la fois par le système et par une librairie au niveau utilisateur. Pour créer des threads, LINUX applique la stratégie du *One-For-One* (une entrée dans la table des processus pour chaque thread).

### 1.1 Les fonctions disponibles

Un extrait des déclarations de `<pthread.h>`

```
// Création d'un thread qui exécute la fonction spécifiée.
// avec l'argument prévu.
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_function) (void *), void *arg);

// Suicide d'un thread.
void pthread_exit(void *return_value);

// Attendre la terminaison d'un thread.
int pthread_join(pthread_t thread, void **return_value);

// Envoyer un signal (UNIX) à un thread.
int pthread_kill(pthread_t thread, int sig);

// Abandonner la CPU pour la donner à un autre thread.
int sched_yield(void);
```

Bien entendu, une aide plus complète peut être obtenue sur chaque fonction en utilisant le manuel UNIX : `man nom_de_fonction`.

### 1.2 Un exemple

Cet exemple est la traduction d'un exercice de TP avec un thread qui lit des caractères au clavier et les passe à un autre thread qui se charge de les afficher. Il faut noter que le thread principal (le père) se charge de la création de ses fils et de l'attente de leur mort. Cette disparition est programmée à l'arrivée du caractère `EOF`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <ctype.h>
#include <assert.h>


/*************************************************************
** Zone d'échange entre les threads:
** La valeur '\0' indique l'absence de donnée.
*************************************************************/
#define EMPTY    ('\0')
volatile int theChar = EMPTY;


/*************************************************************
** Producteur: Lire l'entrée standard et, pour chaque
** caractère, donner le tour au consommateur.
*************************************************************/
void* read_stdin (void* _unused) {
    int nb_chars = 0;
    while (theChar != EOF) {
        while (theChar != EMPTY) { /* attendre mon tour */
            /* rien à faire */
        }
        theChar = getchar();
        nb_chars++;
    }

    /* préparation d'un résultat (à titre d'exemple) */
    int* result = malloc(sizeof(int));
    assert(result != NULL);
    *result = (nb_chars - 1);
    return result;
}


/*************************************************************
** Consommateur: Attendre son tour et, pour chaque caractère,
** l'afficher et donner le tour au producteur.
*************************************************************/
void* write_to_stdout (void* _unused) {
    unsigned long cpt = 0;
    while (true) {
        while (theChar == EMPTY) { /* attendre */
            cpt ++;
        }
        if (theChar == EOF) break;
        theChar = toupper(theChar);
        printf("cpt=%lu, car=%c\n", cpt, theChar);
        theChar = EMPTY;             /* donner le tour */
    }
    return NULL;
}


/*************************************************************
** Créer les threads et attendre leurs terminaisons.
*************************************************************/
int main (void) {
    pthread_t read_thread, write_thread;
    void *nb_chars;

    /* Nous sommes optimistes, mais nous vérifions */
    assert(pthread_create(&read_thread, NULL, read_stdin, NULL) == 0);
    assert(pthread_create(&write_thread, NULL, write_to_stdout, NULL) == 0);
```

**Travail à faire :**

- Compiler et effectuer l'édition de liens avec la ligne suivante :

```
gcc votre_programme.c -pthread -o votre_programme
```

- Dans un autre console, vous pouvez utiliser la commande `ps -eLf` pour observer les *threads* créés par cet exemple.
- Vérifiez la consommation de CPU en utilisant la commande `time votre_programme`.
- Diminuez cette consommation de CPU en utilisant la fonction `usleep(100)` dans les boucles d'attente et vérifiez le résultat (avec la commande `time`).

# 2 Utilisation des sémaphores

La librairie de gestion des *threads* offre les fonctions ci-dessous pour créer et utiliser des sémaphores. **Attention** : ces sémaphores sont propres à un processus. Ils permettent de synchroniser plusieurs threads entre eux, mais ils ne peuvent synchroniser plusieurs processus. Pour réaliser cette synchronisation il faut se tourner vers les sémaphores système V basés sur les IPC (*Inter Processus Communication*).

## 2.1 Définition des fonctions

```
// Création d'un sémaphore et préparation d'une valeur initiale
int sem_init(sem_t *semaphore, int pshared, unsigned int valeur)

// Opération P sur un sémaphore.
int sem_wait(sem_t * semaphore);

// Version non bloquante de l'opération P sur un sémaphore.
int sem_trywait(sem_t * semaphore);

// Opération V sur un sémaphore.
int sem_post(sem_t * semaphore);

// Récupérer le compteur d'un sémaphore.
int sem_getvalue(sem_t * semaphore, int * sval);

// Destruction d'un sémaphore.
int sem_destroy(sem_t * semaphore);
```

**Note :** Une aide plus complète peut être obtenue sur chaque fonction en utilisant le manuel UNIX (`man nom_de_la_fonction`).

## 2.2 Un exemple d'utilisation des sémaphores

Cet exemple illustre la mise en oeuvre d'une section critique (mutuelle exclusion) permettant d'éviter un mélange des affichages réalisés par les deux threads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <assert.h>


sem_t mutex;


/*************************************************************
** Afficher (20 fois) une ligne composée de 40 chiffres.
*************************************************************/
void* affichage (void* ptr_num) {
    int num = *((int*) ptr_num);

    for(int i = 0; i < 20; i++) {
        sem_wait(&mutex); /* prologue */
            for(int j=0; j<20; j++) printf("%d",num);
            sleep(1); /* pour être sur d'avoir des problèmes */
            for(int j=0; j<20; j++) printf("%d",num);
            printf("\n");
            fflush(stdout);
        sem_post(&mutex); /* épilogue */
        sleep(1);
    }
    return NULL;
}


/*************************************************************
** Initialiser le sémaphore, créer les deux threads et
** attendre leurs terminaisons.
*************************************************************/
int main (void) {
    pthread_t th0, th1;
    int num0 = 0, num1 = 1;

    /* ce qui compte est le 1 : valeur initiale du compteur */
    sem_init(&mutex, 0, 1);

    assert(pthread_create(&th0, NULL, affichage, &num0) == 0);
    assert(pthread_create(&th1, NULL, affichage, &num1) == 0);

    assert(pthread_join(th0, NULL) == 0);
    assert(pthread_join(th1, NULL) == 0);

    return (EXIT_SUCCESS);
}
```

**⏩ Travail à faire :**

- Compiler et effectuer l'édition de liens avec la ligne suivante :

```
gcc votre_programme.c -pthread -o votre_programme
```

- Vérifier que le mécanisme d'exclusion mutuelle est bien nécessaire (commentez l'appel des fonctions `sem_wait` et `sem_post`).
- Si vous supprimez l'appel à `sleep` dans la section critique quel est le résultat ? Avez-vous une explication ?

# 3 Sémaphore, producteur et consommateur

## 3.1 Un tampon de un caractère

Reprenez le premier exemple des threads (échange de caractère) et remplacez la boucle d'attente active (sur la valeur `'\0'`) par deux sémaphores pour implanter un schéma producteur consommateur (le tampon est constitué d'un seul caractère). Rappel du schéma producteur/consommateur :

**Initialisation**

```
NPlein : sémaphore := (0, {});
NVide  : sémaphore := (n, {}); // Taille du tampon
```

**Algorithme du producteur :**

```
Pour toujours faire
    P(NVide);
    < produire >
    < déposer dans le tampon >
    V(NPlein);
Fin-pour
```

**Algorithme du consommateur :**

```
Pour toujours faire
    P(NPlein);
    < retirer du tampon >
    < consommer >
    V(NVide);
Fin-pour
```

## 3.2 Un tampon réaliste

Modifiez l'exemple précédent pour avoir maintenant un tampon d'une dizaine de caractères.

```
#define BUFFER_SIZE      (10)
char buffer[ BUFFER_SIZE ];
int ptr_input = 0;
int ptr_output = 0;
```

La variable `ptr_input` donne l'indice de la prochaine case vide du tampon tandis que `ptr_output` donne l'indice de la prochaine case pleine. Ces variables sont gérées comme des pointeurs circulaires.

Prenez soin de ralentir le producteur (avec `sleep`) puis dans un deuxième temps le consommateur pour tester tous les cas de figure (tampon vide ou tampon plein).