# Fiche TP N° 03
# « Manipulation des Processus et Threads en langage C »

**Remarques:**

- Les primitives de manipulation des processus sous Unix/Linux sont :

> **signal()** : gestion des signaux
> **fork()** : création d'un processus
> **pause()** : mise en sommeil sur l'arrivée d'un signal
> **wait()** : mise en sommeil sur la terminaison d'un fils
> **sleep()** : mise en sommeil sur une durée déterminée (argument)
> **kill()** : envoi de signal à un processus
> **exit()** : terminaison d'un processus
>
> **(pour plus d'informations, voir le man de chaque primitive)**

- L'appel `fork()` duplique un processus et le système crée alors une copie complète du processus, avec un PID différent. L'un des deux processus est fils de l'autre.
- `fork()` peut échouer par manque de mémoire ou si l'utilisateur a déjà créé trop de processus; dans ce cas, aucun fils n'est créé et `fork()` retourne -1
- Lors de la création d'un nouveau thread dans un processus, il obtient sa propre pile (et de ce fait ses propres variables locales) mais partage avec son créateur les variables globales.
- Pour compiler un programme en langage C, on utilise soit **gcc** (logiciel libre dans la cadre du projet GNU/GPL) ou bien **cc** (compilateur c/c++)

> **ps [-e][-l] :** Affiche la liste des processus
> - L'option -e : permet d'afficher les processus de tous les utilisateurs.
> - L'option -l : permet d'obtenir plus d'informations dont les plus importantes sont:
>
> (**UID**: identité du propriétaire du processus; – **PID**: numéro du processus; – **PPID**: PID du père du processus; – **NI**: priorité (nice); – **S**: état du processus(**R** si actif, **S** si bloqué, **Z** si terminé).

1- En utilisant l'éditeur **gedit**, écrire les programmes suivants.
2- Compiler et exécuter ces programmes
3- Expliquer le déroulement de ces programmes

**Exercice 1** : gestion des processus par : **`fork(), pid() et ppid()`**

**cprocessus.c**

```
/* Exemple utilisation primitive fork() sous Linux */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void) {
    int pid; /* PID du processus fils */
    int i;
    pid = fork();
    switch (pid) {
    case -1:
        printf("Erreur: echec du fork()\n");
        exit(1);
        break;
```

```
    case 0:
        /* PROCESSUS FILS */
        printf("je suis le processus fils : PID=%d , mon pere est :
        PPID=%d\n", getpid(), getppid());
        exit(0); /* fin du processus fils */
        break;
    default:
        /* PROCESSUS PERE */
        printf("Ici le pere: le fils a un pid=%d\n", pid );
        wait(0); /* attente de la fin du fils */
        printf("Fin du pere.\n");
    }
}
```

**Exercice 2** : gestion des threads à l'aide de l'API POSIX (**pthread.h**)

Ce programme crée un autre thread qui va montrer qu'il partage des variables avec le thread original, et permettre au "petit nouveau" de renvoyer un résultat à l'original.

**N.B:**

- pour compiler ce programme il faut établir un li**cthread.c**liothèque des threads **:**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
void *fonction_de_thread(void *arg);
char message[] = "Hello World";
int main() {
int res;
pthread_t un_thread;
void *resultat_de_thread;
res=pthread_create(&un_thread, NULL, fonction_de_thread, (void*)message);
if (res != 0)
{
perror("Echec de la creation du thread ");
exit(EXIT_FAILURE);
}
printf("En attente de terminaison du thread ...\n");
res=pthread_join (un_thread, &resultat_de_thread);
if (res != 0)
{
perror("Echec de l'ajout de thread ");
exit(EXIT_FAILURE);}
printf("Retour du thread, il a renvoye %s\n",(char *)resultat_de_thread);
printf("Voici a present le message %s\n", message);
exit(EXIT_SUCCESS);
}
void *fonction_de_thread (void *arg)
{
printf("la fonction_de_thread est en cours d'execution. L'argument était :
%s\n", (char *) arg);
sleep (3);
strcpy(message, "Salut !");
pthread_exit("Merci pour le temps processeur");}
```

Adresse de l'identifiant du thread créé

Fonction exécutée par le thread créé

Paramètre passé à la fonction

Identifiant du thread attendu

adresse de l'objet renvoyé au thread appelant

**gcc –o cthread cthread.c –lpthread**

- **pthread_create:** crée un nouveau thread (comme le "fork" pour un processus)
- **pthread_join:** fait attendre la fin d'un thread (comme "wait" pour les processus)
- **pthread_exit :** termine un thread (comme "exit" pour les processus)

# Fiche TP N° 04
# « Threads et Sémaphores avec API POSIX »

**Remarques :**

**Commandes pour les process (processus : programme en exécution)**

**ps [-e][-l] :** Affiche la liste des processus
- L'option -e : permet d'afficher les processus de tous les utilisateurs.
- L'option -l : permet d'obtenir plus d'informations dont les plus importantes sont:

(**UID**: identité du propriétaire du processus; – **PID**: numéro du processus; – **PPID**: PID du père du processus; – **NI**: priorité (nice); – **S**: état du processus(**R** si actif, **S** si bloqué, **Z** si terminé).

1) Pour compiler et effectuer l'édition de liens vous devez utiliser la ligne suivante :

```
gcc –o votre_programme votre_programme.c –lpthread  ou bien
gcc votre_programme.c -lpthread -o votre_programme
```

2) Utilisation de man pour toute aide sur l'utilisation des fonctions ci-après.

## Partie 1: Les threads LINUX (processus de poids légers)

Les threads de LINUX sont gérés à la fois par le système et par une librairie au niveau utilisateur. Voici quelques fonctions standards de la l'API POSIX (API : **A**pplication **P**rogramming **I**nteface et POSIX : **P**ortable **O**perating **S**ystem **I**nterface, dont le **X** exprime l'héritage UNIX de l'API)

```
pthread_create( thread, attribut, routine, argument )
```

Création d'un thread. Le nouveau flot d'exécution démarre en se branchant à la routine spécifiée. Cette routine reçoit l'argument prévu.

```
pthread_exit( résultat )
```

Suicide d'un thread.

```
pthread_join( thread, résultat )
```

Attendre la terminaison d'un autre thread.

```
pthread_kill( thread, nu_du_signal )
```

Envoyer un signal (UNIX) à un thread. C'est un moyen dur pour tuer un thread.

```
sched_yield()
```

Abandonner la CPU pour la donner à un autre thread (ou un autre processus). Attention : il n'existe pas de préemption de la CPU à l'intérieur des threads d'un même processus. En clair, si un thread garde la CPU, les autres threads ne vont pas s'exécuter. Cette routine permet de programmer un partage équitable de la CPU entre threads coopératifs.

**Exercice 1** : gestion des threads à l'aide de l'API POSIX (**pthread.h**)

Un thread 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 "F".

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

volatile char theChar = '\0';
volatile char afficher = 0;

void* lire (void* name) {
    do {
        while (afficher == 1) ; /* attendre mon tour */
        theChar = getchar();
        afficher = 1; /* donner le tour */
        }
    while (theChar != 'F');
    return NULL;
    }

void* affichage (void* name) {
    int cpt = 0;
    do {
        while (afficher == 0) cpt ++; /* attendre */
        printf("cpt = %d, car = %c\n", cpt, theChar);
        afficher = 0; /* donner le tour */
        }
    while (theChar != 'F');
    return NULL;
    }

int main (void) {
    pthread_t filsA, filsB;

    if (pthread_create(&filsA, NULL, affichage, "AA")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_create(&filsB, NULL, lire, "BB")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }

    if (pthread_join(filsA, NULL))
        perror("pthread_join");

    if (pthread_join(filsB, NULL))
        perror("pthread_join");

    printf("Fin du pere\n") ;
    return (EXIT_SUCCESS);
    }
```

## Partie 2: Utilisation des sémaphores pour synchroniser les threads

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*).

```
int sem_init(sem_t *semaphore, int pshared, unsigned int valeur)
```

Création d'un sémaphore et préparation d'une valeur initiale.

```
int sem_wait(sem_t * semaphore);
```

Opération P sur un sémaphore.

```
int sem_trywait(sem_t * semaphore);
```

Version non bloquante de l'opération P sur un sémaphore.

```
int sem_post(sem_t * semaphore);
```

Opération V sur un sémaphore.

```
int sem_getvalue(sem_t * semaphore, int * sval);
```

Récupérer le compteur d'un sémaphore.

```
int sem_destroy(sem_t * semaphore);
```

Destruction d'un sémaphore.

**Exercice 2** : utilisation des sémaphores à l'aide de l'API POSIX (**pthread.h**)

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

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;

void* affichage (void* name) {
    int i, j;
    for(i = 0; i < 20; i++) {
        sem_wait(&mutex); /* prologue */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        sched_yield(); /* pour etre sur d'avoir des problemes */
        for(j=0; j<5; j++) printf("%s ",(char*)name);
        printf("\n ");
        sem_post(&mutex); /* epilogue */
    }
    return NULL;
    }

int main (void) {
    pthread_t filsA, filsB;
    sem_init(&mutex, 0, 1);
    if (pthread_create(&filsA, NULL, affichage, "AA")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_create(&filsB, NULL, affichage, "BB")) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
        }
    if (pthread_join(filsA, NULL))
        perror("pthread_join");
    if (pthread_join(filsB, NULL))
        perror("pthread_join");
    printf("Fin du pere\n") ;
    return (EXIT_SUCCESS);
    }
```

# Fiche TP N° 05
# « Manipulation des Threads en langage Java »

## Remarques:

- Gestion lourde des threads avec C (API POSIX) et simplifiée avec Java.
- En java, les threads sont des instances des classes dérivées (héritées) de la **classe *Thread***
  - ✓ La classe Thread crée des threads généraux, sa méthode **run** ne fait rien.
  - ✓ La méthode **run** indique à un thread les instructions à exécuter
  - ✓ La méthode **run** doit être publique, ne prendre aucun argument, ne renvoyer aucune valeur et ne lever aucune exception.
- Deux techniques pour fournir une méthode run à un thread
  1) hériter la classe Thread (java.lang.Thread) et redéfinir la méthode run.
  2) Implémenter l'interface Runnable (java.lang.Runnable) et définir la méthode run de cette classe
- Un thread commence par exécuter la méthode run de l'objet "cible" qui a été passé au thread.

**Exercice 1** : Technique n°1 : **Hériter la classe Thread**

**Thread1.java**

```
class TPrint extends Thread
{
String txt;
int attente;
public TPrint(String t, int p)
{ txt= t; attente=p;}
public void run (){ for (int  i=0 ; i<8;i++) {
System.out.print (txt+i+" ");
try {
    sleep(attente);
    }
catch (InterruptedException e) {};
}
}
}
public class Thread1 {
static public void main(String args[]){
TPrint a= new TPrint("A", 100); // créer un thread
TPrint b= new TPrint("B", 200); // créer un autre thread
a.start();
b.start();
}
}
```

**N.B:**
Le Résultat des deux exercices devrait être : A0 B0 A1 A2 B1 A3 A4 B2 A5 A6 B3 A7 B4 B5 B6 B7

**Question**: Donner une explication de ce résultat ?

**Exercice 2** : Technique n°2 : **Implémenter l'interface Runnable**

**Thread2.java**

```
import java.io.*;
import java.lang.*;
class TPrint implements Runnable {
String txt;
int attente;
public TPrint (String t, int p)
{ txt= t; attente=p;}
public void run (){ for (int  i=0 ; i<8;i++) {
System.out.print (txt+i+" ");
try {
    Thread.currentThread().sleep(attente);
    }
catch (InterruptedException e) {};
}
}
}
public class Thread2 {
static public void main(String args[]){
TPrint a= new TPrint("A", 100);
TPrint b= new TPrint("B", 200);
new Thread(a).start(); // Créer et lancer un thread
new Thread(b).start();
}
}
```

**Exercice 3** : Soit 2 threads qui comptent de 1 à 10 et à 15. Commençons donc par créer une sous-classe de la classe Thread, puis créons une classe permettant de lancer les deux threads via la méthode main() :

**LanceCompteurs.java**

```
import java.io.*;
import java.lang.*;

class ThreadCompteur extends Thread {
int no_fin;
// Constructeur
ThreadCompteur(int fin) {
no_fin = fin;
}
// On redéfinit la méthode run()
public void run() {
for (int i=1; i<=no_fin ; i++) {
System.out.println(this.getName()+"==>"+i);
}
}
}
// Classe lançant les threads
class LanceCompteurs {
public static void main (String args[]) {
// On instancie les threads
ThreadCompteur cp1 = new ThreadCompteur(10);
ThreadCompteur cp2 = new ThreadCompteur(15);
// On démarre les deux threads
cp1.start();
cp2.start();
```

```
// On attend qu'ils aient fini de compter
while (cp1.isAlive() || cp2.isAlive()) {
// On bloque le thread 100 ms
try {
Thread.sleep(100);
} catch (InterruptedException e) { return; }
}
}
}
```

**Le résultat devrait être similaire à ce qui suit :**

```
Thread-0==>1
Thread-1==>1
Thread-0==>2
Thread-1==>2
Thread-0==>3
Thread-1==>3
Thread-0==>4
Thread-1==>4
Thread-0==>5
Thread-1==>5
Thread-0==>6
Thread-1==>6
Thread-0==>7
Thread-1==>7
Thread-0==>8
Thread-1==>8
Thread-0==>9
Thread-1==>9
Thread-0==>10
Thread-1==>10
Thread-1==>11
Thread-1==>12
Thread-1==>13
Thread-1==>14
Thread-1==>15
```

**Question** : Donner une explication de ce résultat ?

**Rappel :**

1) Pour compiler un programme java (par exemple, exemple.java) :

   **javac exemple.java**

   qui va générer exemple.class (le bytecode de ce programme)

2) Pour l'exécuter :

   **java exemple**

# Fiche TP N° 06
# « Synchronisation des Threads avec les Moniteurs en Java »

Compiler et exécuter les programmes suivants:

**Exercice 1:** TestThread1.java (Attention: Java est sensible à la casse)

```
import java.io.* ;
import java.lang.* ;

class TPrint extends Thread {
        String txt;
        public TPrint(String t) {
              txt = t;
        }
       public void run() {
             for (int j=0; j<3; j++) {
                 for (int i=0; i<txt.length();i++) {
                    System.out.print(txt.charAt(i));
                    try { sleep(100);}
                    catch (InterruptedException e) {};
                    }
             }
       }
}
```

```
public class TestThread1
{
   static public void main(String args[])
   {
    TPrint a = new TPrint("bonjour ");
    TPrint b = new TPrint("au revoir ");
    a.start();
    b.start();
   }
}
```

**// résultat de l'exécution :**

// baoun jroeuvro ibro najuo urre vbooinrj oauur revoir

**Qu'observez-vous?**

**Exercice 2:** TestThread2.java

```
import java.io.* ;
import java.lang.* ;
class MoniteurImpression {
    synchronized public void imprime (String t) {
        for (int i=0; i<t.length(); i++) {
           System.out.print(t.charAt(i));
           try { Thread.currentThread().sleep(100);}
           catch (InterruptedException e) {};
        }
    }
}
class TPrint extends Thread {
     String txt;
    static MoniteurImpression mImp =
                new MoniteurImpression();
    public TPrint(String t) {
          txt = t;
    }
    public void run() {
          for (int j=0; j<3; j++)
                mImp.imprime(txt);
    }
}
```

```
public class TestThread2{
 static public void main(String args[]) {
    TPrint a = new TPrint("bonjour ");
    TPrint b = new TPrint("au revoir ");
    a.start();
    b.start();
  }
}
```

**// resultat de l'execution :**

// bonjour au revoir bonjour au revoir bonjour au revoir

**Qu'observez-vous?**

***Bon Courage***