

# Programmation Système et Réseau



**Par : Anis ZOUAOUI**

**Anis.ZOUAOUI@esprit.ens.tn**

**Anis.zwawi@gmail.com**

# Programmation Système et Réseau

## Chapitre 2

# Les Threads POSIX

Par : Anis ZOUAOUI

# PLAN

- Théorie de threads
  - ✓ Les processus
  - ✓ Les threads
  - ✓ Différents Threads
- Librairie pthread.h
  - ✓ Création
  - ✓ Terminaison
  - ✓ Attente de la fin d'un thread
  - ✓ Nettoyage à la terminaison
  - ✓ Communication entre Threads
  - ✓ TP

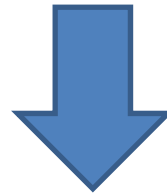
# Théorie de threads(1)

- Processus unix :
  - ✓ Trois segments :
    - Le segment texte : code + données statiques
    - Le segment donnée : variables
    - Le segment stack : Pile du processus
  - ✓ Un environnement
    - Information nécessaire au kernel pour gérer le processus (contenu du processus, priorités, fichiers ouverts ...)
    - Peut être modifié par un appel système

# Théorie de threads(2)

Processus :

- Le modèle processus décrit précédemment est un programme qui s'exécute selon un chemin unique (compteur ordinal).



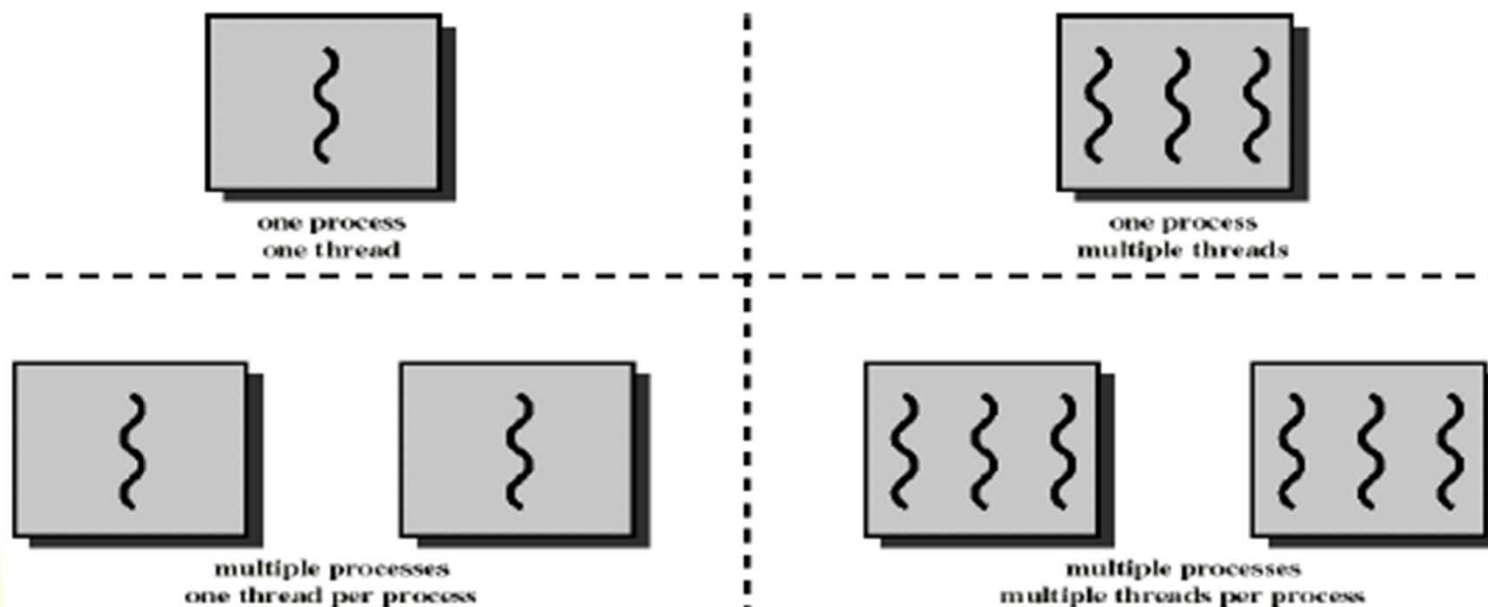
On dit qu'il a un **flot de contrôle unique** (un seul thread).

# Théorie de threads(3)

## Threads :

De nombreux systèmes d'exploitation modernes offrent la possibilité d'associer à un même processus plusieurs chemins d'exécution (multithreading, multiflot d'exécution).

## Processus et threads



# Théorie de threads(4)

Threads :

Un thread est une unité d'exécution rattachée à un processus, chargée d'exécuter une partie du processus.

Un processus est vu comme étant un ensemble de ressources (espace d'adressage, fichiers, périphériques...) que ses threads (flots de contrôle ou processus légers) partagent.

# Théorie de threads(5)

## Threads :

Lorsqu'un processus est créé, un seul flot d'exécution (thread) est associé au processus. Ce thread peut en créer d'autres.

Chaque thread a :

- un identificateur unique
- une pile d'exécution
- des registres (un compteur ordinal)
- un état...

# Comment ça marche?

## Copie de :

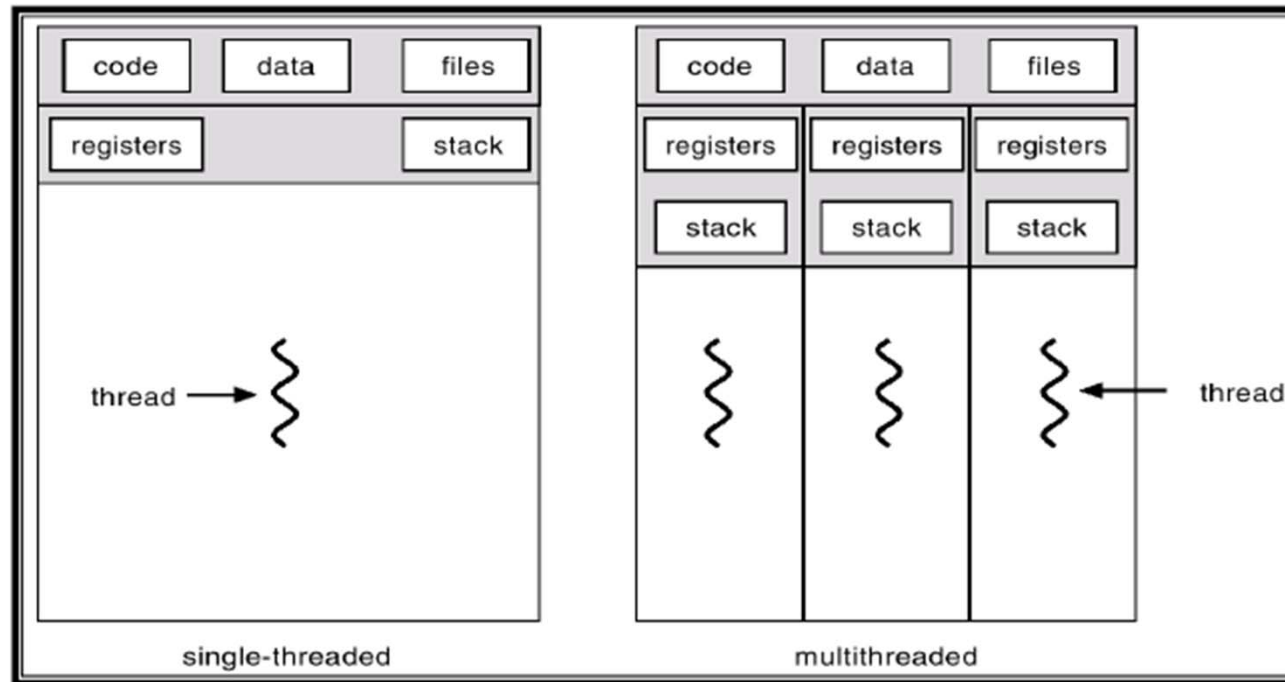
- Compteur ordinal (adresse mémoire de la prochaine instruction à exécuter)
- Registre (emplacement de mémoire interne à un processeur)
- Pile d'appel
- Etat d'ordonnancement (quel processus va prendre la main)

## Partage de :

- Espace d'adressage
- Variables globales
- Fichiers ouverts
- Signaux

# Comment ça marche?

Threads :



Le multithreading permet l'exécution simultanée ou en pseudo-parallèle de plusieurs parties d'un même processus.

# Pourquoi utiliser les threads

- ✓ Améliorer la rapidité des applications (pas de blocages pour des tâches qui peuvent être effectuée en parallèle)
- ✓ Exploiter de manière efficace les machines multiprocesseur
- ✓ Améliorer la structure de votre programme
- ✓ Utiliser moins de ressources système

# Différents Threads :

## ✓ Thread utilisateur :

- Les threads utilisateur sont implantés dans une bibliothèque (niveau utilisateur) qui fournit un support pour les gérer.
- Ils ne sont pas gérés par le noyau.
- Le noyau gère les processus (table des processus) et ne se préoccupe pas de l'existence des threads (**modèle plusieurs-à-un**).
- Lorsque le noyau alloue le processeur à un processus, le **temps d'allocation du processeur est réparti entre les différents threads du processus** (cette répartition n'est pas gérée par le noyau).

# Différents Threads :

## ✓ Thread Noyau:

- Les threads noyau sont **directement supportés par le système d'exploitation**.
- Le système d'exploitation se charge de leur gestion. **Du temps CPU est alloué à chaque thread**. (modèle un-à-un)
- Si un thread d'un processus est **bloqué**, un autre thread du **même processus** peut être **élu** par le noyau
- Cette implémentation est plus intéressante pour les systèmes **multiprocesseurs**.

# Threads POSIX : Librairie pthread.h

## Création d'un thread :

```
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,  
void*(*start_routine)(void *), void *arg);  
// renvoie 0 si l'appel réussit, sinon !=0 : identifiant de l'erreur
```

## Explication :

pthread\_t \*tid : ID du thread

const pthread\_attr\_t \*tattr : les attributs du thread  
(taille de la pile, priorité....)

void\*(\*start\_routine)(void \*) : La fonction à exécuter

void \*arg : le paramètre de la fonction

# Threads POSIX : Librairie pthread.h

Exemple :

```
#include <pthread.h>
void* thr_f (void* param)
{
    int * t = (int *) param;
    printf ("parametre : %d", *t);
}
int main ()
{
    pthread_t thr1;
    pthread_create(&thr1,NULL,thr_f,arg);
    return 0;
}
```

Pour compiler ajouter l'option **-lpthread**

# Threads POSIX : Librairie pthread.h

## Exercice :

Ecrire un programme qui lance deux thread, l'un affichant 50 fois 1, et l'autre 50 fois 2.

# Threads POSIX : Librairie pthread.h

Attendre qu'un thread se termine :

```
int pthread_join(thread_t tid, void **status);
```

**Explication :**

Suspend l'activité de la thread appelante tant que la thread tid n'est pas terminée.

**status** contient la valeur de retour du thread tid lors de sa terminaison.

# Threads POSIX : Librairie pthread.h

Terminer une thread:

```
void pthread_exit(void *status);
```

## Explication :

Termine l'exécution la thread courante avec une valeur de retour particulière

# Threads POSIX : Librairie pthread.h

**Avoir le TID d'un thread :**

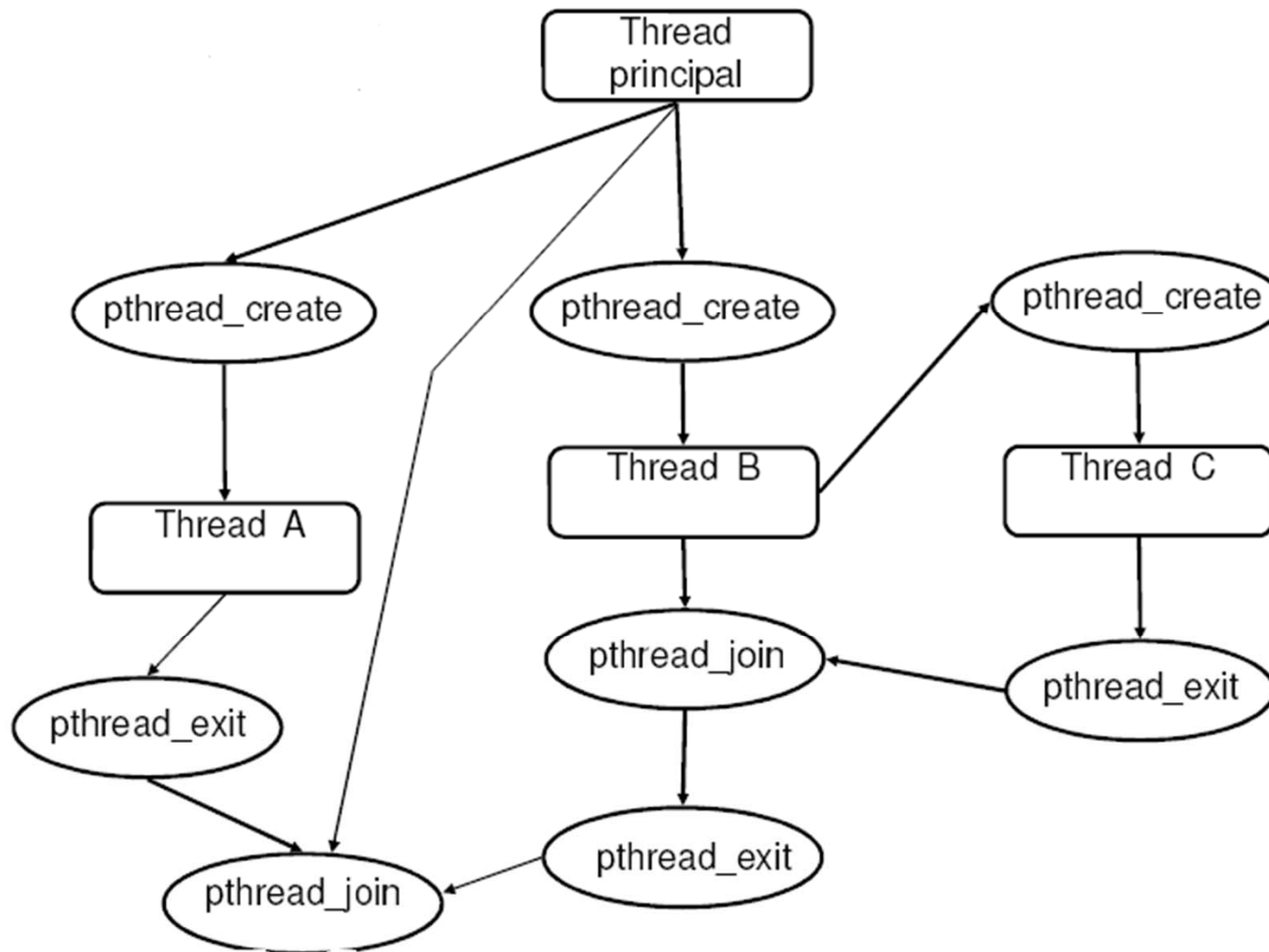
```
pthread_t pthread_self(void);
```

**Explication :**

Retourne le TID d'un thread.

# Threads POSIX : Librairie pthread.h

Exemple :



# Threads POSIX : Librairie pthread.h

## Exemple :

```
// exemple_threads.c
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void afficher(int n, char lettre)
{
    int i,j;
    for (j=1; j<n; j++)
    {
        printf("%c",lettre);
        fflush(stdout);
    }
}
void *threadA()
{ afficher(100,'A');
  printf("\n Fin du thread A\n");
  fflush(stdout);
  pthread_exit(NULL);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple :

```
void *threadC(void *inutilise)
{
    afficher(150,'C');
    printf("\n Fin du thread C\n");
    fflush(stdout);
    pthread_exit(NULL);
}

void *threadB(void *inutilise)
{
    pthread_t thC;
    pthread_create(&thC, NULL, threadC, NULL);
    afficher(100,'B');
    printf("\n Le thread B attend la fin du thread
    C\n");
    pthread_join(thC,NULL);
    printf("\n Fin du thread B\n");
    fflush(stdout);
    pthread_exit(NULL);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple :

```
int main()
{
    int i;
    pthread_t thA, thB;
    printf("Creation du thread A");
    pthread_create(&thA, NULL, threadA, NULL);
    pthread_create(&thB, NULL, threadB, NULL);
    sleep(1);
    //attendre la fin des threads
    printf("Le thread principal attend que les autres se
    terminent\n");
    pthread_join(thA,NULL);
    pthread_join(thB,NULL);
    exit(0);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 2 : Partage de variable

```
// programme threads.c
#include <unistd.h> //pour sleep
#include <pthread.h>
#include <stdio.h>
int glob=0;
void* decrement(void * x)
{
    glob = glob - 1 ;
    printf("ici decrement[%u], glob = %d\n",
    pthread_self(),glob);
    pthread_exit(NULL);
}
void* increment (void * x)
{
    sleep(1);
    glob = glob + 1;
    printf("ici increment[%u], glob = %d\n", pthread_self(),
    glob);
    pthread_exit(NULL);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 2 : Partage de variable

```
int main( )
{
    pthread_t tid1, tid2;
    printf("ici main[%d], glob = %d\n", getpid(),glob);
    //création d'un thread pour incrément
    if ( pthread_create(&tid1, NULL, increment, NULL) != 0)
        return -1;
    printf("ici main: creation du thread[%u] avec succes\n",tid1);
    // creation d'un thread pour decrement
    if ( pthread_create(&tid2, NULL, decrement, NULL) != 0)
        return -1;
    printf("ici main: creation du thread [%u] avec succes\n",tid2);
    pthread_join(tid1,NULL); // attendre la fin d'un thread
    pthread_join(tid2,NULL);
    printf("ici main : fin des threads, glob = %d \n",glob);
    return 0;
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 3 : Passage de paramètres à un thread

```
// Programme p11.1.cpp
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <pthread.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
using namespace std;

int MAX=5;
inline int gen_alea( int, int );
void *afficher( void * );
```

# Threads POSIX : Librairie pthread.h

## Exemple 3 : Passage de paramètres à un thread

```
int main(int argc, char *argv[])
{
    pthread_t thread_id[MAX];
    int retour;
    setvbuf(stdout, (char *) NULL, _IONBF, 0);
    if ( argc > MAX+1 ){ // verifier la liste d'arguments
        cerr << *argv << " arg1, arg2, ... arg" << MAX << endl;
        return 1;
    }
    cout << "Affichage" << endl;
    for (int i = 0; i < argc-1; ++i)
    { // création des threads
        if( pthread_create(&thread_id[i],NULL,afficher, (void *)argv[i+1]) > 0)
        {
            cerr << "Echec a la création des threads" << endl;
            return 2;
        }
    }
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 3 : Passage de paramètres à un thread

```
for (int i=0; i < argc-1; ++i)
{ // attendre les threads
  if ( pthread_join(thread_id[i], (void **) retour) > 0){
    cerr << "Echec de l'attente des threads" << endl;
    return 3;
  }
  cout << endl << "Thread " << thread_id[i] << " retourne " << retour;
}
cout << endl << "Termine" << endl;
return 0;
// fin de main()
}
```

# Threads POSIX : Librairie pthread.h

## Thread et mutex :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Déclarer le verrou **static**

Déclarer la variable partagée **volatile**

# Threads POSIX : Librairie pthread.h

## Exemple 4 : Thread et mutex

Imaginons un simple tableau d'entier rempli par un thread (lent) et lu par un autre (plus rapide). Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu. Pour cela, on peut utiliser les *MUTEX* afin de protéger le tableau pendant le temps de son remplissage:

# Threads POSIX : Librairie pthread.h

## Exemple 4 : Thread et mutex

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

static pthread_mutex_t my_mutex;
static int tab[5];

void *read_tab_process (void * arg)
{ int i;
  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++)
    printf ("read_process, tab[%d] vaut %d\n", i, tab[i]);
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 4 : Thread et mutex

```
void *write_tab_process (void * arg)
{ int i;
  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++) {
    tab[i] = 2 * i;
    printf ("write_process, tab[%d] vaut %d\n", i, tab[i]);
    sleep (1); /* Relentit le thread d'ecriture... */ }
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 4 : Thread et mutex

```
main (int ac, char **av)
{
    pthread_t th1, th2;
    void *ret;
    pthread_mutex_init (&my_mutex, NULL);
    if (pthread_create (&th1, NULL, write_tab_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 1\n");
        exit (1); }
    if (pthread_create (&th2, NULL, read_tab_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 2\n");
        exit (1); }
    (void)pthread_join (th1, &ret);
    (void)pthread_join (th2, &ret);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 5 : Thread et Sémaphore

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

static sem_t my_sem;
int the_end;

void *thread1_process (void * arg)
{ while (!the_end) {
    printf ("Je t'attend !\n");
    sem_wait (&my_sem);
  }
  printf ("OK, je sors !\n");
  pthread_exit (0);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 5 : Thread et Sémaphore

```
void *thread2_process (void * arg)
{ register int i;
  for (i = 0 ; i < 5 ; i++) {
    printf ("J'arrive %d !\n", i);
    sem_post (&my_sem);
    sleep (1); }
  the_end = 1;
  sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
  pthread_exit (0);
}
```

# Threads POSIX : Librairie pthread.h

## Exemple 5 : Thread et Sémaphore

```
main (int ac, char **av)
{
    pthread_t th1, th2;
    void *ret;
    sem_init (&my_sem, 0, 0);
    if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 1\n");
        exit (1); }
    if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 2\n");
        exit (1); }
    (void)pthread_join (th1, &ret);
    (void)pthread_join (th2, &ret);
}
```

# Threads POSIX : Librairie pthread.h

## Exercice :

Ecrire un programme contenant deux thread. L'un remplit un tableau de 100 cases, et l'autre le lit. Dans un premier temps, le remplissage se fait sans rendre le verrou, et dans un autre temps, le remplissage est ralenti (sleep) et rend le verrou.

# Threads POSIX : Librairie pthread.h

## Attributs d'un thread :

le type d'ordonnancement

la priorité

la taille de la pile

## Par défaut :

SCHED\_OTHER : toutes les threads se partagent le processeur en fonction de leur priorité.

Hérite de la priorité des parents

Taille de la pile par défaut : 61440

# Threads POSIX : Librairie pthread.h

## Modifier les attributs d'un thread :

### Modifier l'ordonnancement :

```
int pthread_attr_setsched(pthread_attr_t *attr_pt, int scheduler);
```

### Modifier la priorité :

```
int pthread_attr_setprio(pthread_attr_t *attr_pt, int prio);
```

### Modifier la taille de la pile :

```
int pthread_attr_setstacksize(pthread_attr_t *attr_pt, long size);
```

# Threads POSIX : Librairie pthread.h

## Exercice :

Ecrire un programme qui lance deux thread, l'un affichant 50 fois 1, et l'autre 50 fois 2. Ce coup-ci, le second thread est prioritaire (priorité= SCHED\_FIFO).