Chapitre V : Threads

Cours Système Exploitation II

1 Linfo

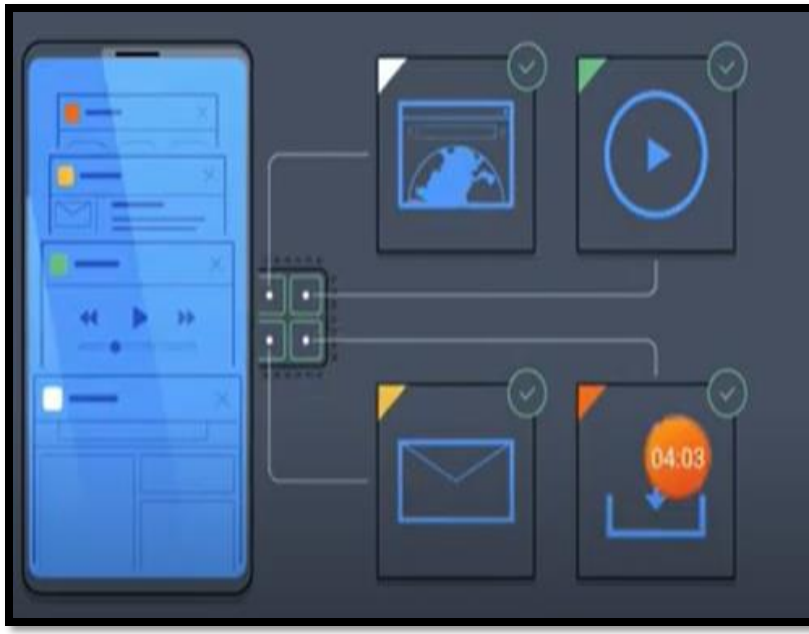
2024/2025

Dr Sana BENZARTI **ISIMM**

Introduction

- Programme multitâche de lancer plusieurs parties de son code en même temps.
- À chaque partie du code sera associé un sous-processus pour permettre l'exécution en parallèle.

- Exemple : ouvrir plusieurs onglets d'un navigateur : un onglet pour lire un article en ligne, un autre pour regarder youtube et un autre pour utiliser ChaGPT.



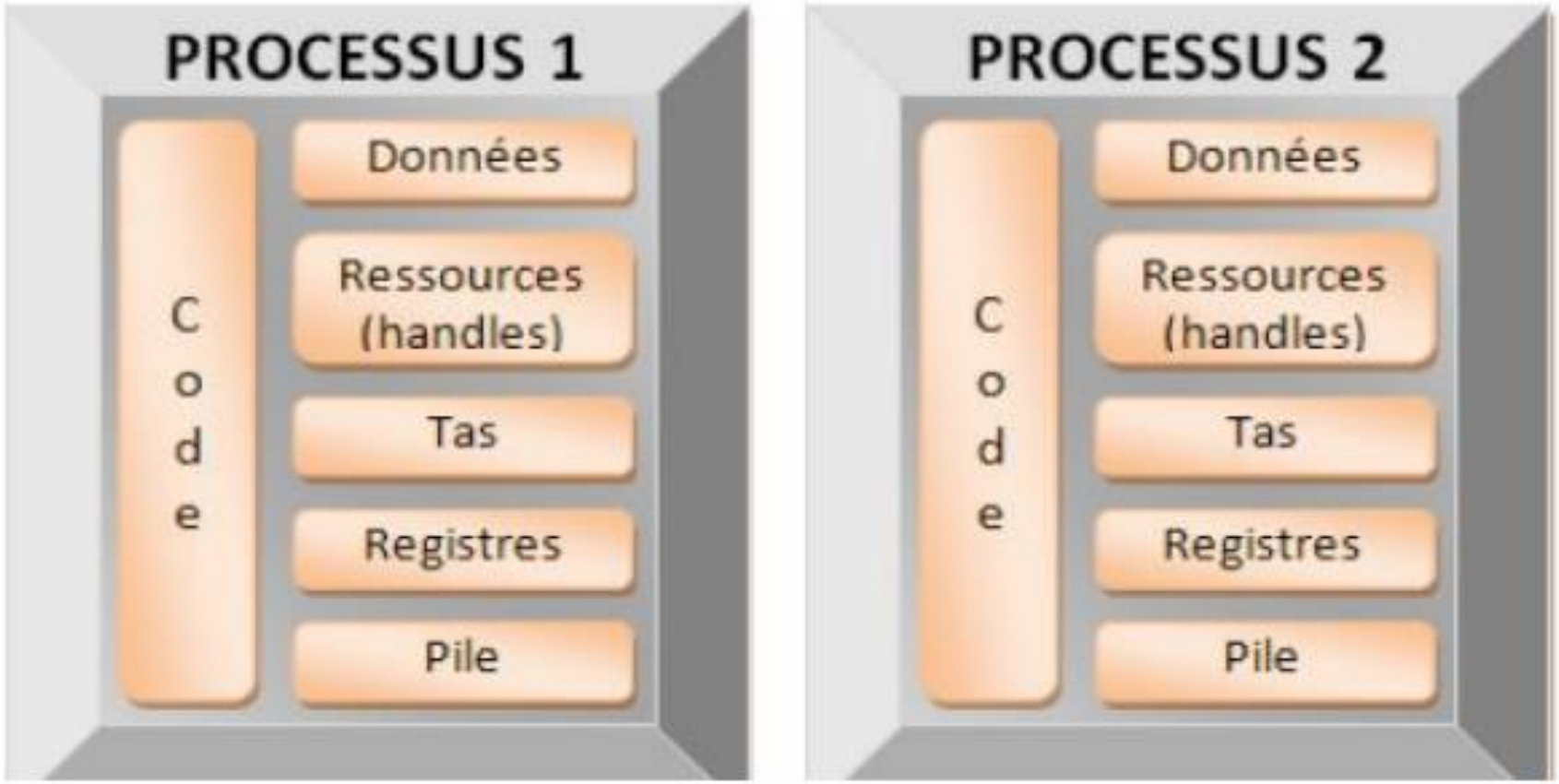
- Processus coûte cher au lancement: espace mémoire, la commutation de contexte, la communication inter-processus (échange d'informations) seront lourdes à gérer

Introduction

- Peut-on avoir un sous-processus qui permettrait de lancer une partie du code d'une application sans qu'il soit onéreux?

- **Les threads qui sont appelés aussi « processus légers ».**

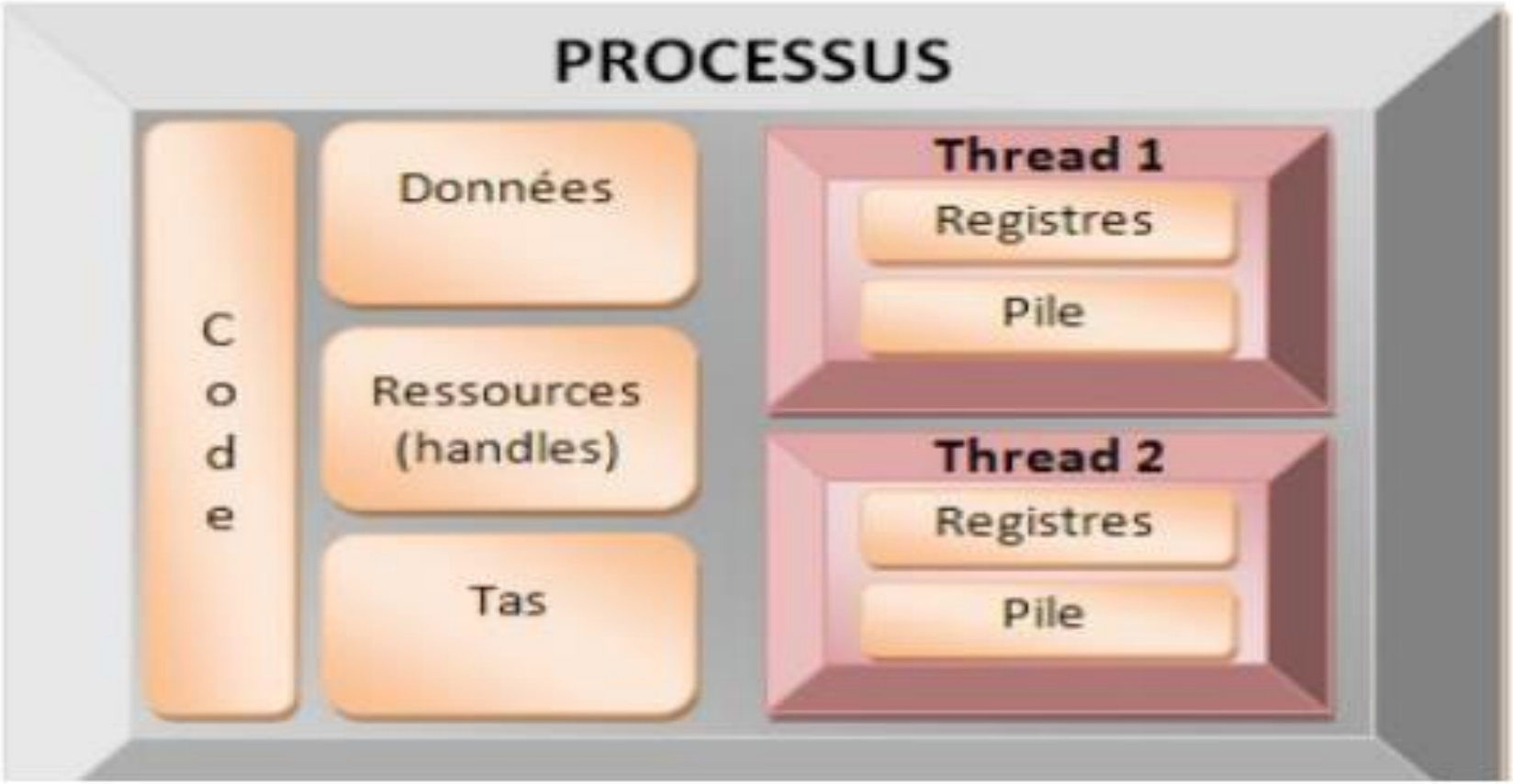
  - **Threads = tâches = fil d'execution**



« Les processus lourds » sont complètement isolés les uns des autres car ils ont chacun leur contexte.

L'exécution d'un processus est réalisée de manière isolée par rapport aux autres processus.

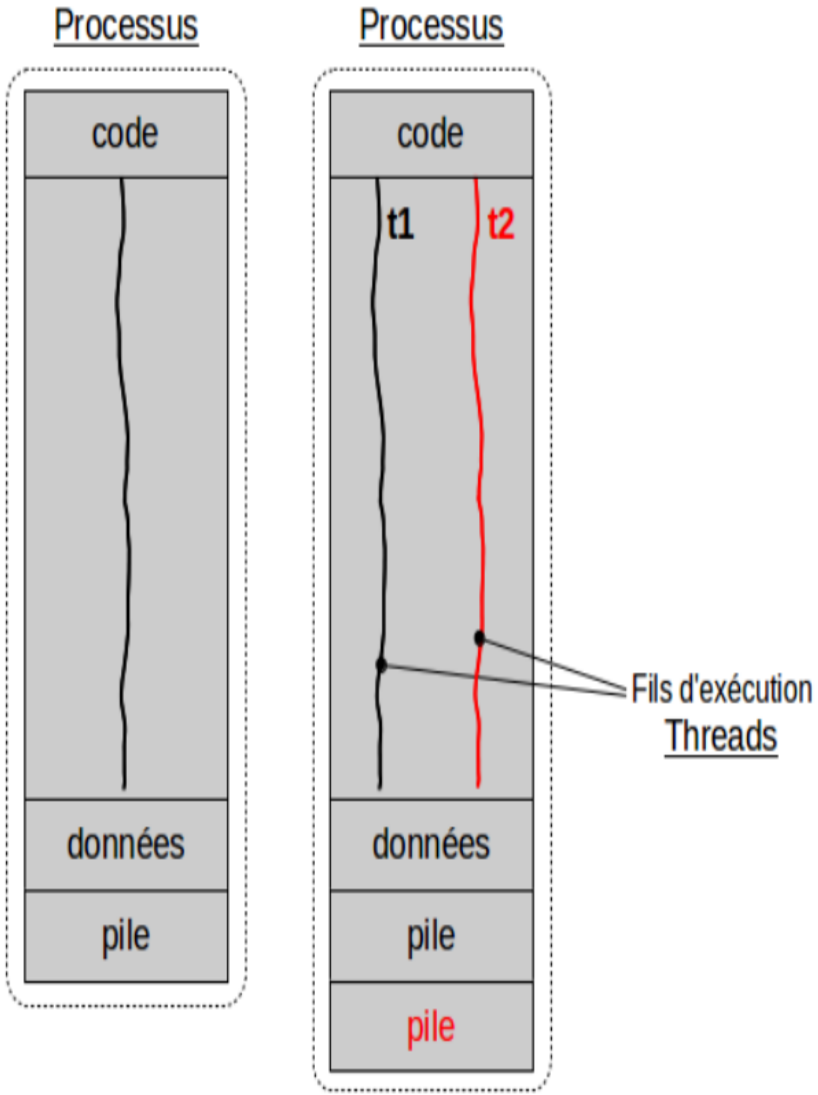
Données	Code, pile, tas, données statiques/globales
Registres	PC, SP, registres généraux, flags (contexte du CPU)
Ressources	Fichiers, sockets, IPC, signaux, PID, etc.



« Les processus légers » (threads) partagent un contexte commun sauf la pile (les threads possèdent leur propre pile d'appel).

Cas pratiques des threads

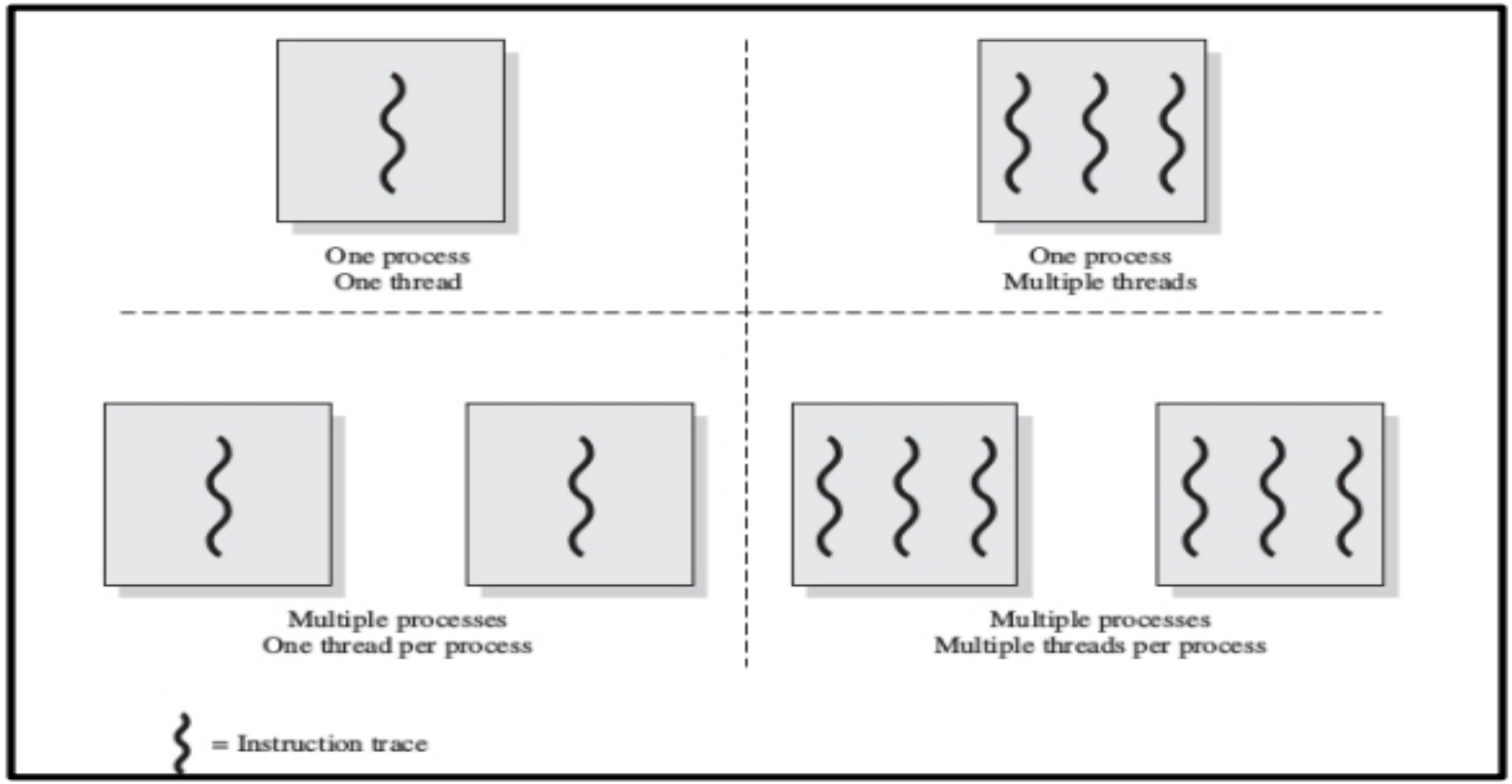
Cas pratique	Description
Jeux vidéo	Un thread pour l'affichage, un pour la physique, un pour le son, un pour l'IA...
Serveurs Web	Un thread par client connecté (Apache, Nginx en mode worker)
Applications mobiles	Gérer le téléchargement de données sans bloquer l'interface
Applications scientifiques	Exécuter plusieurs calculs en parallèle
IDE / éditeurs de code	Compilation en tâche de fond pendant qu'on écrit du code
Traitement d'image / vidéo	Appliquer des filtres ou encoder plusieurs morceaux en parallèle
Surveillance de capteurs	Un thread par capteur pour lire les données en temps réel



un processus léger, lorsqu'il est créé, partage le même espace mémoire et la même table des fichiers ouverts que son processus (son créateur), mais dispose de sa propre pile.

L'avantage des processus légers sur les processus lourds est qu'ils sont très favorables à la programmation multitâche, car la création d'une tâche est moins coûteuse en termes de ressources du fait qu'elles ne sont pas toutes dupliquées.

Monothread vs Multithread



Monothread | **Multithread**

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

thread : pointeur de type « pthread_t » contenant l'identificateur de la tâche qui vient d'être créée ;

attr : variable de type « pthread_attr_t ». Elle correspond à une sorte de conteneur qui va permettre d'indiquer les différentes propriétés de la tâche qui doit être exécutée (son type d'ordonnancement, sa priorité, tâche joignable/détachable? ...). (on va se contenter a la mettre a NULL)

start_routine : c'est la fonction C qui sera exécutée par la tâche qui est créée ;

arg : pointeur correspond aux variables passées en paramètre à la fonction « start_routine ». Il vaut « NULL » si aucun paramètre n'est passé à la fonction ;

Création d'un thread

- On déclare d'abord une instance de pthread :

```
pthread_t ta; // ta une instance de pthread
```

- Pour créer un thread, il faudra utiliser la fonction « **pthread_create** ». La signature d'une telle fonction est comme suit :

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

Création et terminaison d'un thread

- Cette fonction retourne « 0 » si le thread a été créé.

Un pointeur vers une instance de « pthread ».
Ce pointeur contient l'identificateur du thread créé

« attr » contient les attributs du thread à créer. On passe la valeur « NULL » pour initialiser les attributs avec les valeurs par défaut.

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

« arg » est l'argument que nous allons passer à la fonction « start_routine ».

Une fonction que le thread va exécuter. Cette fonction retourne un « void* » et accepte un argument du type « void* ».

Identifiant d’un pthread

- Un pthread n’a pas de PID propre (ils partagent tous le même PID, celui du processus contenant les threads)

- Chaque thread possède un identifiant unique de type pthread_t (équivalent du pid_t, c’est une structure opaque)

- Pthread_t pthread_self() : retourne l’ID du thread (équivalent à getpid())

- Pthread_equal(pthread_t a, pthread b) permet de comparer deux identifiants

Attente d'un thread

```
int pthread_join(pthread_t thread, void **
    retval);
```

`retval` : un pointeur sur une variable (contenant un pointeur), où le code retour sera copié.

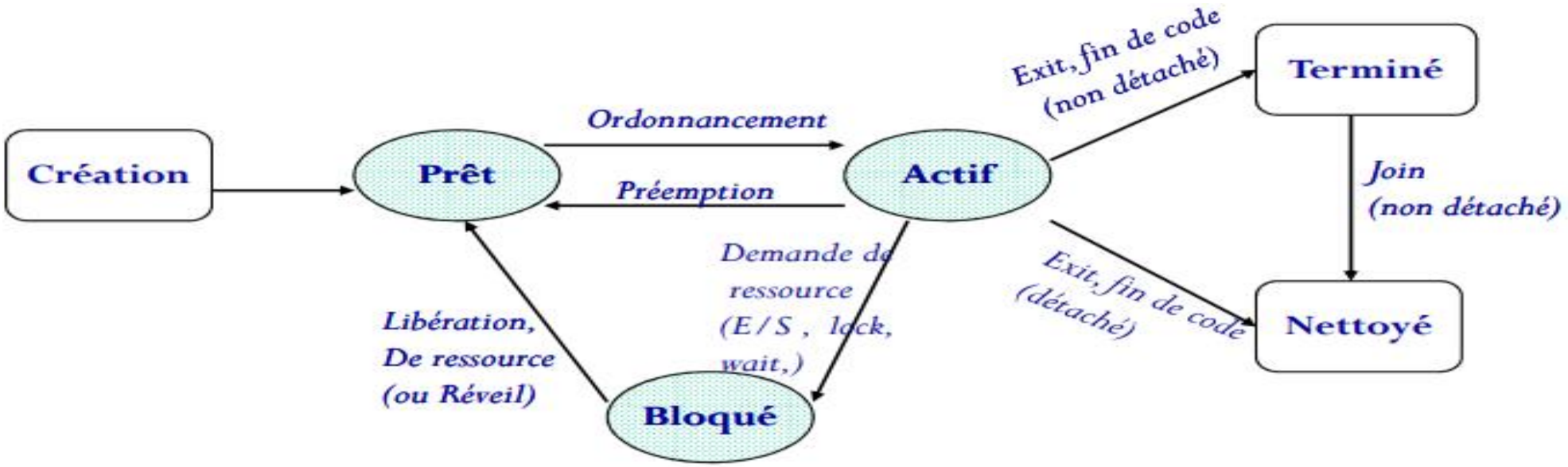
- NULL : ignoré
- thread "annulé" : `PTHREAD_CANCELED`

Terminaison d'un thread

```
int pthread_exit(void *retval);
```

Arrête (termine) le thread courant.
retval sera la valeur retour (récupérée par pthread_join)

Cycle de vie d'un thread



Si on ne veut pas avoir à gérer la fin d'un thread, on peut le "détacher".

```
int pthread_detach(pthread_t thread);
```

- La structure sera détruite à la terminaison du thread
- Il ne sera pas possible de retrouver son pointeur retour avec `pthread_join`

- La fonction « pthread_detach » est utilisée pour détacher un thread. Quand ce thread prendra fin, l'espace mémoire occupé par celui-ci sera libéré.

- En le mettant dans un état détaché, nous n'avons pas besoin d'attendre qu'un autre thread fasse un appel à pthread_join.

```
int pthread_detach(pthread_t th);
```

Pas **de pthread_join**, le **thread peut terminer son exécution, mais ses ressources ne seront pas récupérées** par le programme principal.

Résultat : **"ressources suspendues"** → cela ressemble à un **thread zombie**, bien que techniquement ce soit un concept différent des processus zombies.

Un thread terminé mais non "recolté" est comme un **processus zombie**, car il a terminé mais on ne l'a pas "nettoyé".

pthread_detach()

- Cela dit au système que **le thread se nettoiera tout seul** une fois fini.

Pas de thread zombie ici, car les ressources sont libérées automatiquement.

Et s'il n'y a ni pthread_join ni pthread_detach ?

On risque d’avoir une **fuite de ressources** (resource leak) pour chaque thread terminé, car **le système garde ses informations** en mémoire pour un éventuel join.

Action	Effet
pthread_join()	Attend la fin du thread, récupère ses ressources
pthread_detach()	Libère automatiquement les ressources à la fin
Aucun des deux	Le thread terminé reste dans un état suspendu → fuite mémoire possible (style zombie)

Différents types de
Passage d'arguments avec la fonction pthread_create()

Ne rien passer (NULL)

```
void *thread_func(void *arg) {
    printf("Thread lancé sans argument\n");
    pthread_exit(NULL);
}

pthread_create(&t, NULL, thread_func, NULL);
```

Passer un entier (via pointeur)

```
void *thread_func(void *arg) {
    int *n = (int *)arg;
    printf("Valeur reçue : %d\n", *n);
    pthread_exit(NULL);
}

int val = 10;
pthread_create(&t, NULL, thread_func, (void *)&val);
```

Passer un tableau (par exemple `int[]`)

```
void *thread_func(void *arg) {
    int *tab = (int *)arg;
    printf("Tab[0] = %d, Tab[1] = %d\n", tab[0], tab[1]);
    pthread_exit(NULL);
}

int tab[] = {5, 10};
pthread_create(&t, NULL, thread_func, (void *)tab);
```

Passer une structure

```
typedef struct {
    int id;
    char nom[20];
} Info;
```

Passer une structure

```
void *thread_func(void *arg) {
    Info *info = (Info *)arg;
    printf("ID: %d, Nom: %s\n", info->id, info->nom);
    pthread_exit(NULL);
}

Info data = {1, "Alice"};
pthread_create(&t, NULL, thread_func, (void *)&data);
```

Passer une chaîne de caractères (char *)

```
void *thread_func(void *arg) {
    char *msg = (char *)arg;
    printf("Message reçu : %s\n", msg);
    pthread_exit(NULL);
}

char *message = "Bonjour";
pthread_create(&t, NULL, thread_func, (void *)message);
```

```
typedef struct Node {
    int valeur;
    struct Node* suivant;
} Node;
```

```
Node* liste = malloc(sizeof(Node));
liste->valeur = 42;
liste->suivant = NULL;
```

On passe la liste directement, car c’est **déjà un pointeur** :

pthread_create(&thread, NULL, afficher_liste, **(void *)liste**);

```
void* afficher_liste(void* arg) {
    Node* courant = (Node*) arg; // on récupère le
pointeur vers la tête

    while (courant != NULL) {
        printf("Valeur : %d\n", courant->valeur);
        courant = courant->suivant;
    }

    return NULL;
}
```

Exercices d'applications

Exercice 1 : Course entre deux threads

Créer deux threads représentant deux coureurs :
Le coureur A affiche "A court !" toutes les 200ms
Le coureur B affiche "B court !" toutes les 500ms
Le thread principal doit attendre que les deux threads aient fini.
Chaque coureur court 5 fois.

Utiliser pthread_create et pthread_join.

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

void *coureurA(void *arg) {
    for (int i = 0; i < 5; i++) {
        printf(" Coureur A court !\n");
        usleep(200000); // 200 ms
    }
    pthread_exit(NULL);
}
```

```
void *coureurB(void *arg) {
    for (int i = 0; i < 5; i++) {
        printf(" Coureur B court !\n");
        usleep(500000); // 500 ms
    }
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, coureurA, NULL);
    pthread_create(&t2, NULL, coureurB, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Course terminée !\n");
    return 0;
}
```

Exercice 2

Détacher un thread

Créer un thread qui affiche un message au bout de 2 secondes. Le thread principal ne doit **pas attendre** ce thread : il continue et affiche "Fin du programme principal".

pthread_detach pour que le thread se détache.

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

void *message_secret(void *arg) {
    sleep(2);
    printf(" Message secret : Tu as bien compris les threads !\n");
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t t;

    pthread_create(&t, NULL, message_secret, NULL);
    pthread_detach(t); // On ne veut pas attendre ce thread

            printf("Fin du programme principal (sans attendre le message)\n");

    sleep(3); // Pour laisser le temps au message d'apparaître
    return 0;
}
```

Exercice 3 :

Mini-jeu du lancer de dé multi-thread

Vous avez 4 joueurs (threads). Chaque joueur lance un dé et affiche sa valeur aléatoire entre 1 et 6.
Le thread principal attend tous les joueurs avant d'annoncer la fin.

Utiliser pthread_create, pthread_join, et rand().

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

#define NB_JOUEURS 4

void *joueur(void *arg) {
    int id = *((int *)arg);
    int lancer = rand() % 6 + 1;
    printf(" Joueur %d a lancé : %d\n", id, lancer);
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t threads[NB_JOUEURS];
    int ids[NB_JOUEURS];

    srand(time(NULL)); // Initialisation de rand()

    for (int i = 0; i < NB_JOUEURS; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, joueur, &ids[i]);
    }
```

```
  for (int i = 0; i < NB_JOUEURS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf(" Tous les joueurs ont lancé le dé !\n");
    return 0;
}
```

Exercice 4

- Ecrire un programme qui créé deux threads pour exécuter chacun deux fonctions .

- Le premier thread affiche des étoiles et le second des dièses

Exercice

Solution 1

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

void *afficher_etoiles(void *arg) {
   int i;
   for (i = 0; i < 10; i++) {
      printf("* ");
   }
   printf("\n");
   pthread_exit(NULL);
}

void *afficher_dieses(void *arg) {
   int i;
   for (i = 0; i < 10; i++) {
      printf("# ");
   }
   printf("\n");
   pthread_exit(NULL);
}
```

Exercice

```
int main() {
   pthread_t thread1, thread2;
   int res1, res2;

   res1 = pthread_create(&thread1, NULL, afficher_etoiles, NULL);
   if (res1 != 0) {
      printf("Erreur lors de la création du thread 1\n");
      exit(EXIT_FAILURE);
   }

   res2 = pthread_create(&thread2, NULL, afficher_dieses, NULL);
   if (res2 != 0) {
      printf("Erreur lors de la création du thread 2\n");
      exit(EXIT_FAILURE);
   }

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   return 0;
}
```

Solution 2

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

// Fonction pour le thread qui affiche des étoiles '*'
void* afficher_etoiles(void* arg) {
    char c = *(char*)arg;
    for (int i = 0; i < 50; i++) {
        printf("%c", c);
        fflush(stdout); // Forcer l'affichage immédiat
    }
    printf("\n");
    return NULL;
}

// Fonction pour le thread qui affiche des dièses '#'
void* afficher_dieses(void* arg) {
    char c = *(char*)arg;
    for (int i = 0; i < 50; i++) {
        printf("%c", c);
        fflush(stdout);
    }
    printf("\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    char etoile = '*';
    char diese = '#';

    // Création de deux threads avec deux fonctions différentes
    pthread_create(&thread1, NULL, afficher_etoiles, &etoile);
    pthread_create(&thread2, NULL, afficher_dieses, &diese);

    // Attente de la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
```

Solution 3

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

// Fonction pour le thread qui affiche des étoiles '*'
void* afficher_etoiles(void* arg) {
    char *msg = (char*)arg;
    {
        printf("%s", msg);

    }

    return NULL;
}

// Fonction pour le thread qui affiche des dièses '#'
void* afficher_dieses(void* arg) {
    char *msg1 = (char*)arg;

        printf("%s", msg1);

    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    char *etoile = "******";
    char *diese = "#####";

    // Création de deux threads avec deux fonctions différentes
    pthread_create(&thread1, NULL, afficher_etoiles, (void*)etoile);
    pthread_create(&thread2, NULL, afficher_dieses, (void*)diese);

    // Attente de la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Fin du programme principal.\n");
    return 0;
}
```

Solution 4

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

// Fonction pour le thread qui affiche des étoiles '*'
void* afficher_etoiles(void* arg) {
    char *msg = (char*)arg;
    {
        printf("%s", msg);

    }

    return NULL;
}

// Fonction pour le thread qui affiche des dièses '#'
void* afficher_dieses(void* arg) {
    char *msg1 = (char*)arg;

        printf("%s", msg1);

    return NULL;
}

int main() {
    pthread_t thread1, thread2;



    // Création de deux threads avec deux fonctions différentes
    pthread_create(&thread1, NULL, afficher_etoiles, "******");
    pthread_create(&thread2, NULL, afficher_dieses, "#######");

    // Attente de la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Fin du programme principal.\n");
    return 0;
}
```