

Faculté des sciences – Center d'excellence

Filière d'excellence : Ingénierie Informatique et Systèmes Embarqués (IISE)

Module : Programmation temps réel

Compte Rendu

TP1 : Programmation Temps Réel

Tâches, Thread POSIX.

Réalisé par :

GOUSSA Oussama

Professeur :

OUKDACH Yassine

Année universitaire 2023/2024

Exercice 1 :

1. Le facteur d'utilisation du processeur.

Le facteur d'utilisation du processeur, est une mesure de l'activité du processeur. Il indique la proportion du temps pendant laquelle le processeur est occupé à exécuter des tâches. Il se calcule comme suit :

$$U = \frac{C}{P}$$

Où :

C : la durée d'exécution, calculée en temps processeur. Il s'agit du pire temps d'exécution.

P : sa période, pour le cas d'une tâche périodique.

Pour T_0 :

$$U_0 = \frac{C}{P} = \frac{2}{6} = \frac{1}{3} \approx 0,33$$

Pour T_1 :

$$U_1 = \frac{C}{P} = \frac{3}{8} \approx 0,37$$

Pour T_2 :

$$U_2 = \frac{C}{P} = \frac{4}{24} = \frac{1}{6} \approx 0,16$$

2. Le facteur de charge du processeur.

Le facteur de charge du processeur est une mesure de l'utilisation du processeur. Il indique le nombre moyen de tâches en cours d'exécution à un moment donné. Il se calcule comme suit :

$$ch = \frac{C}{D}$$

Où :

C : la durée d'exécution, calculée en temps processeur. Il s'agit du pire temps d'exécution.

D : la délai critique, au-delà duquel le résultat est jugé comme étant non pertinent.

Parce que $D = P$

Alors

Pour T_0 :

$$ch_0 = U_0 = \frac{1}{3} \approx 0,33$$

Pour T_1 :

$$ch_1 = U_1 = \frac{3}{8} \approx 0,37$$

Pour T₂ :

$$ch_2 = U_1 = \frac{1}{6} \approx 0,16$$

3. Le temps de réponse.

Le temps de réponse **TR_i** d'une tâche **T_i** se définit comme la différence entre le temps de fin **f_i** de la tâche et le temps de début **r_i** de son exécution. Il représente le temps total que la tâche **T_i** a passé dans le système, en incluant le temps d'attente avant son exécution et le temps d'exécution lui-même. Le calcul du temps de réponse **TR_i** d'une tâche s'effectue comme suit :

$$TR_i = f_i - r_i$$

Où :

TR_i : est le temps de réponse de la tâche T_i

f_i : est le temps de fin de la tâche T_i

r_i : est le temps de début de l'exécution de la tâche T_i

Pour T₀ :

$$TR_0 = f_0 - r_0 = 2 - 0 = 2 \text{ s}$$

$$TR_0 = f_1 - r_1 = 8 - 6 = 2 \text{ s}$$

$$TR_0 = f_2 - r_2 = 14 - 12 = 2 \text{ s}$$

$$TR_0 = f_3 - r_3 = 20 - 18 = 2 \text{ s}$$

Pour T₁ :

$$TR_1 = f_0 - r_0 = 5 - 0 = 5 \text{ s}$$

$$TR_1 = f_1 - r_1 = 11 - 8 = 3 \text{ s}$$

$$TR_1 = f_2 - r_2 = 21 - 16 = 5 \text{ s}$$

Pour T₂ :

$$TR_2 = f_0 - r_0 = 16 - 0 = 16 \text{ s}$$

4. La laxité nominale.

La laxité nominale : Indique le retard maximum que peut prendre la tâche sans dépasser son échéance. Il se calcule comme suit :

$$L = D - C$$

Où :

D : la délai critique, au-delà duquel le résultat est jugé comme étant non pertinent.

C : la durée d'exécution, calculée en temps processeur. Il s'agit du pire temps d'exécution.

Pour T₀ :

$$L_0 = D - C = 5 - 2 = 4$$

Pour T_1 :

$$L_1 = D - C = 8 - 3 = 5$$

Pour T_2 :

$$L_2 = D - C = 24 - 4 = 20$$

5. La gigue de release relative, la gigue de release absolue, la gigue de fin relative et la gigue de fin absolue.

Gigue de release relative (Relative Release Jitter) : d'une tâche est la déviation maximale des temps de démarrage de deux instances consécutives.

Pour T_0 :

$$RRJ_0 = \max | (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) |$$

J = 1 :

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (6 - 6) - (0 - 0) | = 0$$

J = 2 :

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (12 - 12) - (6 - 6) | = 0$$

J = 3 :

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (18 - 18) - (12 - 12) | = 0$$

$$\Rightarrow RRJ_0 = 0$$

Pour T_1 :

$$RRJ_1 = \max | (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) |$$

J = 1 :

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (8 - 8) - (2 - 0) | = 2$$

J = 2 :

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (16 - 16) - (8 - 8) | = 0$$

$$\Rightarrow RRJ_1 = 2$$

Pour T_2 :

$$RRJ_2 = \max | (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) |$$

$$| (S_{i,j+1} - r_{i,j+1}) - (S_{i,j} - r_{i,j}) | = | (5 - 0) | = 5$$

$$\Rightarrow RRJ_2 = 5$$

Gigue de release absolue (Absolute Release Jitter) : d'une tâche est la déviation maximale des temps de départ sur toutes les instances :

Pour T_0 :

$$ARJ_0 = \max (S_{i,j} - r_{i,j}) - \min (S_{i,j} - r_{i,j})$$

J = 1 :

$$s_{i,j} - r_{i,j} = 0 - 0 = 0$$

J = 2 :

$$s_{i,j} - r_{i,j} = 6 - 6 = 0$$

J = 3 :

$$s_{i,j} - r_{i,j} = 12 - 12 = 0$$

J = 4 :

$$s_{i,j} - r_{i,j} = 18 - 18 = 0$$

On a $s_{i,j} - r_{i,j} = \{0, 0, 0, 0\}$

Donc :

$$\Rightarrow ARJ_0 = 0 - 0 = 0$$

Pour T₁ :

$$ARJ_1 = \max (s_{i,j} - r_{i,j}) - \min (s_{i,j} - r_{i,j})$$

J = 1 :

$$s_{i,j} - r_{i,j} = 2 - 0 = 2$$

J = 2 :

$$s_{i,j} - r_{i,j} = 8 - 8 = 0$$

J = 3 :

$$s_{i,j} - r_{i,j} = 16 - 16 = 0$$

On a $s_{i,j} - r_{i,j} = \{2, 0, 0\}$

Donc :

$$\Rightarrow ARJ_1 = 2 - 0 = 2$$

Pour T₂ :

$$ARJ_2 = \max (s_{i,j} - r_{i,j}) - \min (s_{i,j} - r_{i,j})$$

$$s_{i,j} - r_{i,j} = 5$$

$$\Rightarrow ARJ_2 = 5 - 0 = 5$$

Gigue de fin relative (Relative Finishing Jitter) : d'une tâche est la déviation maximale des temps de fin de deux instances consécutives :

Pour T₀ :

$$RFJ_0 = \max | (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) |$$

J = 1 :

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (8 - 6) - (2 - 0) | = 0$$

J = 2 :

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (14 - 12) - (8 - 6) | = 0$$

J = 3 :

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (20 - 18) - (14 - 12) | = 0$$

$$\Rightarrow RFJ_0 = 0$$

Pour T₁ :

$$RFJ_1 = \max | (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) |$$

J = 1 :

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (11 - 8) - (5 - 0) | = 2$$

J = 2 :

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (21 - 16) - (11 - 8) | = | 5 - 3 | = 2$$

$$\Rightarrow RFJ_1 = 2$$

Pour T₂ :

$$RFJ_2 = \max | (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) |$$

$$| (f_{i,j+1} - r_{i,j+1}) - (f_{i,j} - r_{i,j}) | = | (16 - 6) | = 10$$

$$\Rightarrow RFJ_2 = 10$$

Gigue de fin absolue (Absolute Finishing Jitter) : d'une tâche est la déviation maximale des temps de fin sur toutes les instances :

Pour T₀ :

$$AFJ_0 = \max (f_{i,j} - r_{i,j}) - \min (f_{i,j} - r_{i,j})$$

J = 1 :

$$f_{i,j} - r_{i,j} = 2 - 0 = 2$$

J = 2 :

$$f_{i,j} - r_{i,j} = 8 - 6 = 2$$

J = 3 :

$$f_{i,j} - r_{i,j} = 14 - 12 = 2$$

J = 4 :

$$f_{i,j} - r_{i,j} = 20 - 18 = 2$$

On a $f_{i,j} - r_{i,j} = \{2, 2, 2, 2\}$

Donc :

$$\Rightarrow AFJ_0 = 2 - 2 = 0$$

Pour T₁ :

$$AFJ_1 = \max (f_{i,j} - r_{i,j}) - \min (f_{i,j} - r_{i,j})$$

J = 1 :

$$f_{i,j} - r_{i,j} = 5 - 0 = 5$$

J = 2 :

$$f_{i,j} - r_{i,j} = 11 - 8 = 3$$

J = 3 :

$$f_{i,j} - r_{i,j} = 21 - 16 = 5$$

On a $f_{i,j} - r_{i,j} = \{5, 3, 5\}$

Donc :

$$\Rightarrow AFJ_1 = 5 - 3 = 2$$

Pour T₂ :

$$AFJ_2 = \max (f_{i,j} - r_{i,j}) - \min (f_{i,j} - r_{i,j}) = 16 - 0 = 16$$

$$\Rightarrow ARJ_2 = 16$$

Exercice 2 :

Ce code crée un thread qui exécute la fonction **print_message**, affichant le message passé en argument, puis le programme principal (main) attend la fin de l'exécution du thread avec **pthread_join**.

- La fonction `print_message` prend un pointeur vers une chaîne de caractères en argument et affiche ce message à l'écran.
- Dans la fonction `main`, une chaîne de caractères `message` contenant le message à afficher est créée.
- La fonction `pthread_create` est utilisée pour créer un thread en lui passant la fonction `print_message` et le message comme arguments.
- Le programme principal (`main`) attend la fin de l'exécution du thread avec `pthread_join`, ce qui garantit que le message est affiché avant la fin du programme.
- Le programme est compilé avec l'option `-pthread` pour inclure la bibliothèque `pthread`, qui est nécessaire pour utiliser les fonctionnalités de threading en C.
- Enfin, le programme est exécuté, et le message "Bonjour, OUSSAMA" est affiché par le thread.

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

// Fonction pour afficher un message à l'écran
void *print_message(void *arg) {
    // Cast de l'argument vers une chaîne de caractères
    char *message;
    message = (char *)arg;

    // Affichage du message
    printf("%s \n", message);

    // Le thread se termine, retourne NULL
    return NULL;
}
```

```

int main(int argc, char *argv[]) {
    pthread_t thread; // Déclaration d'un identifiant de thread

    const char *message = "Bonjour, OUSSAMA"; // Message à afficher

    // Création du thread en lui passant la fonction print_message
    et le message comme argument
    pthread_create(&thread, NULL, print_message, (void *)message);

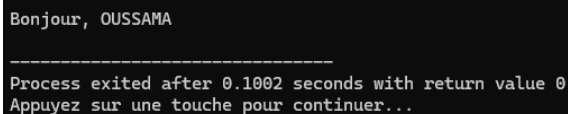
    // Attendre la fin de l'exécution du thread
    pthread_join(thread, NULL);

    return EXIT_SUCCESS;
}

```

Exécution du programme :

À l'exécution, le message "Bonjour, OUSSAMA" sera affiché à l'écran par le thread créé.



```

Bonjour, OUSSAMA
-----
Process exited after 0.1002 seconds with return value 0
Appuyez sur une touche pour continuer...

```

Exercice 3 :

1. Exécuter le premier programme et donner le résultat de son exécution. Qu'est ce que vous remarquer ?

```

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

// Fonction exécutée par le premier thread
void *Tache1(void *arg) {
    int i = 0;
    while (i < 5) { // Répéter 5 fois
        printf("Execution de Tache1\n"); // Afficher un message
        sleep(1); // Attendre 1 seconde
        i++;
    }
    return NULL;
}

// Fonction exécutée par le deuxième thread
void *Tache2(void *arg) {
    int j = 0;
    while (j < 3) { // Répéter 3 fois
        printf("Execution de Tache2\n"); // Afficher un message
        sleep(1); // Attendre 1 seconde
        j++;
    }
}

```



```

    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2; // Déclaration des identifiants de
    thread


    // Création des deux threads pour les tâches Tache1 et Tache2
    pthread_create(&thread1, NULL, Tache1, NULL);
    pthread_create(&thread2, NULL, Tache2, NULL);

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

    return EXIT_SUCCESS;
}

```

Exécution du programme :



```

Execution de Tache1
Execution de Tache2
Execution de Tache1
Execution de Tache2
Execution de Tache2
Execution de Tache1
Execution de Tache1
Execution de Tache1
Execution de Tache1

Process returned 0 (0x0)   execution time : 5.050 s
Press any key to continue.

```

Le premier programme crée deux threads, un pour chaque tâche (Tache1 et Tache2). Ces tâches sont conçues pour s'exécuter pendant un certain nombre d'itérations en affichant un message toutes les secondes. Les deux tâches sont exécutées en parallèle, ce qui signifie que les messages de chaque tâche peuvent apparaître dans un ordre différent à chaque exécution du programme. La fonction `pthread_join` est utilisée pour attendre la fin de l'exécution de chaque thread avant de terminer le programme.

2. Exécuter le deuxième programme et donner le résultat de son exécution. Qu'est ce que vous remarquer ?

```

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

// Fonction exécutée par le premier thread
void *Tache1(void *arg) {
    int i = 0;
    while (i < 5) { // Répéter 5 fois
        printf("Execution de Tache1\n"); // Afficher un message
        sleep(1); // Attendre 1 seconde
        i++;
    }
}

```

```

    }
    return NULL;
}

// Fonction exécutée par le deuxième thread
void *Tache2(void *arg) {
    int j = 0;
    while (j < 3) { // Répéter 3 fois
        printf("Execution de Tache2\n"); // Afficher un message
        sleep(1); // Attendre 1 seconde
        j++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2; // Déclaration des identifiants de
    thread

    // Création du premier thread pour la tâche Tache1
    pthread_create(&thread1, NULL, Tache1, NULL);

    // Attente de la fin de l'exécution du premier thread
    pthread_join(thread1, NULL);

    // Création du deuxième thread pour la tâche Tache2
    pthread_create(&thread2, NULL, Tache2, NULL);

    // Attente de la fin de l'exécution du deuxième thread
    pthread_join(thread2, NULL);

    return EXIT_SUCCESS;
}

```

Exécution du programme :

```

Execution de Tache1
Execution de Tache1
Execution de Tache1
Execution de Tache1
Execution de Tache1
Execution de Tache2
Execution de Tache2
Execution de Tache2
-----
Process exited after 8.101 seconds with return value 0
Appuyez sur une touche pour continuer...

```

Dans le deuxième programme, le thread pour Tache1 est créé en premier, et le programme attend ensuite que ce thread se termine avant de créer et d'attendre le thread pour Tache2. Ainsi, toutes les itérations de Tache1 sont exécutées avant même que Tache2 ne commence.

Cela donne l'impression que Tache2 est exécutée après Tache1 est terminée, même si en réalité, les deux tâches sont conçues pour s'exécuter en parallèle.

3. Expliquer la différence entre les deux résultats.

Premier programme :

Dans le premier programme, les deux threads, Tache1 et Tache2, s'exécutent en parallèle. Cela signifie qu'ils s'exécutent simultanément, sans se bloquer l'un l'autre.

Par conséquent, l'impression des messages "**Execution de Tache1**" et "**Execution de Tache2**" s'entremêle, résultant en un ordre non-déterminé.

Deuxième programme :

Dans le deuxième programme, l'exécution de Tache2 est retardée jusqu'à la fin de l'exécution de Tache1.

Cela s'explique par l'appel à `pthread_join(thread1, NULL)` ; avant la création de `thread2`. Cette instruction bloque le thread principal jusqu'à la terminaison de Tache1, ce qui signifie que Tache2 ne peut pas commencer avant que Tache1 ait terminé son exécution.

Par conséquent, dans le deuxième programme, l'impression des messages suit un ordre strict : d'abord tous les messages "**Execution de Tache1**", puis tous les messages "**Execution de Tache2**".

Exercice 4 :

Inclusion des bibliothèques nécessaires :

- `stdio.h` pour les entrées/sorties standard.
- `stdlib.h` pour la gestion de la mémoire dynamique.
- `pthread.h` pour la gestion des threads POSIX.

Déclaration des fonctions des threads :

- `thread_func1(void *arg)` : fonction exécutée par le premier thread. Elle affiche le message "Thread 1 : Bonjour !" et retourne `NULL` pour indiquer la fin de son exécution.
- `thread_func2(void *arg)` : fonction exécutée par le deuxième thread. Elle affiche le message "Thread 2 : Salut !" et retourne `NULL` pour indiquer la fin de son exécution.

Fonction `main` :

- Déclaration de deux variables de type `pthread_t` pour identifier les threads : `thread1` et `thread2`.
- Création des threads :
 - `pthread_create(&thread1, NULL, thread_func1, NULL)` : crée le premier thread et lui attribue la fonction `thread_func1`. Le paramètre `NULL` indique qu'aucun attribut spécifique n'est défini pour le thread.
 - `pthread_create(&thread2, NULL, thread_func2, NULL)` : crée le deuxième thread et lui attribue la fonction `thread_func2`. Le paramètre `NULL` indique qu'aucun attribut spécifique n'est défini pour le thread.

- Attente de la fin des threads :
 - `pthread_join(thread1, NULL)` : attend la fin du premier thread identifié par `thread1`. Le paramètre `NULL` indique qu'on ne s'intéresse pas au retour de la fonction du thread.
 - `pthread_join(thread2, NULL)` : attend la fin du deuxième thread identifié par `thread2`. Le paramètre `NULL` indique qu'on ne s'intéresse pas au retour de la fonction du thread.
- Affichage d'un message indiquant la fin du programme.

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

// Première fonction à exécuter dans le premier thread
void *thread_func1(void *arg) {
    // Affichage du message du premier thread
    printf("Thread 1: Bonjour !\n");

    // Le thread se termine, retourne NULL
    return NULL;
}

// Deuxième fonction à exécuter dans le deuxième thread
void *thread_func2(void *arg) {
    // Affichage du message du deuxième thread
    printf("Thread 2: Salut !\n");

    // Le thread se termine, retourne NULL
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2; // Déclaration des identifiants de
    thread

    // Création des deux threads en leur passant les fonctions
    correspondantes
    pthread_create(&thread1, NULL, thread_func1, NULL);
    pthread_create(&thread2, NULL, thread_func2, NULL);

    // Attente de la fin de l'exécution des deux threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return EXIT_SUCCESS;
}
```

Exécution du programme :

```
Tread1: Bonjour !
Tread2: Salut !

-----
Process exited after 0.2108 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Exercice 5 :

Le code crée et exécute plusieurs tâches périodiques à l'aide de threads. Les tâches s'exécutent à des intervalles définis et affichent un message lorsqu'elles sont exécutées. Après 10 secondes, les threads sont annulés et le programme se termine.

Structure `PeriodicTask` et fonction `taskFunction`:

- La structure `PeriodicTask` permet de définir une tâche périodique. Elle comporte deux champs :
 - `id` : Un identifiant unique pour la tâche.
 - `period` : La période d'exécution de la tâche en secondes.
- La fonction `taskFunction` est exécutée par chaque thread de tâche périodique. Elle effectue les actions suivantes :
 - Attend la période spécifiée (`period`) en utilisant la fonction `sleep`.
 - Affiche un message indiquant que la tâche a été exécutée.
 - Est conçue pour s'exécuter indéfiniment tant que le thread n'est pas annulé.

Création et exécution des tâches périodiques :

- Dans la fonction `main`, un tableau `tasks` est créé pour stocker les tâches périodiques.
- Chaque élément du tableau `tasks` est une instance de la structure `PeriodicTask` représentant une tâche individuelle.
- Pour chaque tâche, un thread est créé à l'aide de la fonction `pthread_create`.
 - Ce thread exécute la fonction `taskFunction`.
 - L'adresse de la tâche correspondante est passée à `taskFunction` comme argument.

Annulation et terminaison des threads :

- Après 10 secondes d'exécution, les threads sont annulés à l'aide de la fonction `pthread_cancel`.
- La fonction `pthread_join` est ensuite utilisée pour attendre la terminaison de chaque thread.
 - Cela garantit que tous les threads ont été correctement terminés avant la fin du programme.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // Pour utiliser la fonction sleep

#define NUM_TASKS 5

// Structure pour représenter une tâche périodique
```

```

typedef struct {
    int id;        // Identifiant de la tâche
    int period;    // Période d'exécution de la tâche
} PeriodicTask;

// Fonction exécutée par chaque tâche périodique
void *taskFunction(void *arg) {
    // Conversion de l'argument en pointeur vers une structure
    PeriodicTask
    PeriodicTask *task = (PeriodicTask *) arg;

    int ancien_etat;
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ancien_etat); //
    Activation de l'annulation

    while (1) {
        // Attente de la période spécifiée avant la prochaine
        exécution
        sleep(task->period);

        // Affichage d'un message indiquant que la tâche est
        exécutée
        printf("Tâche %d exécutée\n", task->id);

        // Vérification de la demande d'annulation
        pthread_testcancel();
    }
    return NULL;
}

int main() {
    int i;

    // Définition des tâches périodiques dans un tableau
    PeriodicTask tasks[] = {
        {1, 1},
        {2, 9},
        {3, 3},
        {4, 7},
        {5, 5}
    };

    // Création d'un thread pour chaque tâche
    pthread_t threads[NUM_TASKS];

    for (i = 0; i < NUM_TASKS; i++) {
        pthread_create(&threads[i], NULL, taskFunction, &tasks[i]);
    }

    // Attente 10 secondes avant d'annuler les threads
    sleep(10);
}

```

```

// Annulation des threads après 10 secondes
for (i = 0; i < NUM_TASKS; i++) {
    pthread_cancel(threads[i]);
    pthread_join(threads[i], NULL);
}

return EXIT_SUCCESS;
}

```

Exécution du programme :

```

Tâche 1 exécutée
Tâche 1 exécutée
Tâche 3 exécutée
Tâche 1 exécutée
Tâche 1 exécutée
Tâche 5 exécutée
Tâche 1 exécutée
Tâche 3 exécutée
Tâche 1 exécutée
Tâche 4 exécutée
Tâche 1 exécutée
Tâche 1 exécutée
Tâche 2 exécutée
Tâche 3 exécutée
Tâche 1 exécutée
Tâche 5 exécutée
Tâche 1 exécutée
Tâche 3 exécutée
Tâche 4 exécutée
Tâche 5 exécutée
Tâche 3 exécutée
Tâche 2 exécutée
Tâche 3 exécutée
Tâche 5 exécutée
Tâche 4 exécutée
Tâche 5 exécutée

-----
Process exited after 25.09 seconds with return value 0
Appuyez sur une touche pour continuer... |

```

Exercice 6 :

Ce code démontre efficacement l'utilisation des pthreads pour le calcul parallèle en répartissant la tâche de calcul de la somme du tableau entre plusieurs threads. Le mutex garantit un accès synchronisé à la variable partagée, empêchant la corruption des données et assurant des résultats précis.

Includes:

- `stdio.h` pour les entrées/sorties standard.
- `stdlib.h` pour la gestion de la mémoire dynamique.
- `pthread.h` pour la création et la gestion des threads.

Définitions:

- `ARRAY_SIZE` définit la taille du tableau à 10.
- `NUM_THREADS` définit le nombre de threads à utiliser (4 dans ce cas).

Variable globale:

- `totalSum` est une variable partagée pour stocker la somme totale calculée par tous les threads.

Structure **PartialSum**:

- Cette structure contient les informations nécessaires pour le calcul partiel d'un sous-tableau :
 - `start`: un pointeur vers le début du sous-tableau.
 - `end`: un pointeur vers la fin du sous-tableau.
 - `lock`: un verrou `pthread_mutex_t` pour synchroniser l'accès à la variable partagée `totalSum`.

Fonction **sum_partial**:

- Cette fonction est exécutée par chaque thread.
- Elle calcule la somme partielle des éléments du sous-tableau qui lui est attribué.
- Elle utilise un verrou pour protéger la variable partagée `totalSum` lors de la mise à jour.

Fonction **main**:

- Crée un tableau `array` d'entiers et l'initialise avec des valeurs.
 - Initialise un verrou `pthread_mutex_t lock`.
 - Déclare des tableaux `threads` et `thread` pour stocker les identifiants des threads et les structures `PartialSum`, respectivement.
 - Calcule la taille de chaque sous-tableau (`taille`).
 - Crée chaque thread :
 - Initialise la structure `PartialSum` pour le thread actuel.
 - Crée le thread en appelant `pthread_create` et lui passe la fonction `sum_partial` et la structure `PartialSum` comme arguments.
 - Attend la fin de l'exécution de tous les threads en utilisant `pthread_join`.
 - Affiche la somme totale `totalSum`.
 - Détruit le verrou `lock`.
-
- Crée et synchronise les threads, en passant à chacun la structure `PartialSum` correspondante.
 - Calcule la somme totale en additionnant les sommes partielles.
 - Affiche la somme totale.
 - Détruit le mutex.

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

#define ARRAY_SIZE 10    // Taille du tableau
#define NUM_THREADS 4    // Nombre de threads

int totalSum = 0;        // Variable pour stocker la somme totale

// Structure pour contenir les informations nécessaires pour
// effectuer le calcul partiel
typedef struct {
    int *start;           // Pointeur vers le début du sous-tableau
    int *end;             // Pointeur vers la fin du sous-tableau
```



```

    pthread_mutex_t *lock; // Verrou pour synchroniser l'accès à la
variable partagée
} PartialSum;

// Fonction exécutée par chaque thread pour calculer la somme
partielle
void *sum_partial(void *args) {
    PartialSum *partial = (PartialSum *)args; // Conversion de
l'argument en pointeur de type PartialSum

    int partialSum = 0; // Variable pour stocker la somme partielle

    // Parcours du sous-tableau et calcul de la somme partielle
    int *p;
    for (p = partial->start; p < partial->end; p++) {
        partialSum += *p;
    }

    // Verrouillage pour synchroniser l'accès à la variable partagée
    pthread_mutex_lock(partial->lock);

    // Mise à jour de la somme totale avec la somme partielle
calculée par ce thread
    totalSum += partialSum;

    // Déverrouillage du verrou
    pthread_mutex_unlock(partial->lock);

    // Fin du thread
    pthread_exit(NULL);
}

int main() {
    int array[ARRAY_SIZE]; // Déclaration du tableau
    int i;

    // Initialisation du tableau avec des valeurs de votre choix
    for (i = 0; i < ARRAY_SIZE; ++i) {
        array[i] = i + 1;
    }

    pthread_mutex_t lock; // Déclaration du verrou
    pthread_mutex_init(&lock, NULL); // Initialisation du verrou

    pthread_t threads[NUM_THREADS]; // Déclaration des identifiants
de threads
    PartialSum thread[NUM_THREADS]; // Déclaration des structures
PartialSum pour chaque thread

    int taille = ARRAY_SIZE / NUM_THREADS; // Calcul de la taille de
chaque sous-tableau

```

```

    // Création des threads et division du travail entre eux
    for (i = 0; i < NUM_THREADS; ++i) {
        // Remplissage des informations de la structure PartialSum
        // pour le thread actuel
        thread[i].start = (array + i * taille); // Pointeur vers le
        début du sous-tableau
        thread[i].end = (array + ((i == NUM_THREADS - 1) ?
        ARRAY_SIZE : (i + 1) * taille)); // Pointeur vers la fin du sous-
        tableau
        thread[i].lock = &lock; // Passage du verrou

        // Création du thread et appel de la fonction sum_partial
        avec les informations de la structure actuelle
        pthread_create(&threads[i], NULL, sum_partial, (void
        *)&thread[i]);
    }

    // Attente de la fin de l'exécution de tous les threads
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }

    // Affichage de la somme totale
    printf("Somme totale : %d\n", totalSum);

    // Destruction du verrou
    pthread_mutex_destroy(&lock);

    return 0; // Fin du programme
}

```

Exécution du programme :

```

Somme totale : 55
Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.

```