

COMPTE RENDU

Département : informatique

Filière d'excellence : Ingénierie informatique et systèmes embarqués

Effectué du : 27/04/2024

Sous le thème :

TP1 : PROGRAMMATION TEMPS REEL

Réalisé par :

-SARMA FATIMA ZAHRA

-LAHSUNI NORA

Encadré par :

- M. OUKDACH YASSINE

Année Universitaire

2023/2024

EXERCICE 1 :

Calculer pour chaque tâches:

- 1) le facteur d'utilisation du processeur

$$U = C/P$$

$$T0: U = 2/6 = 0.333$$

$$T1: U = 3/8 = 0.375$$

$$T2: U = 4/24 = 0.166$$

- 2) le facteur de charge du processeur

$$Ch = C/D$$

on a $D = P$ car cette tâche ($T0, T1, T2$) est un tâche périodique à échéance sur requête

$$T0: ch = 2/6 = 0.333$$

$$T1: ch = 3/8 = 0.375$$

$$T2: ch = 4/24 = 0.166$$

- 3) le temps de réponse.

$$TR_i = f_i - r_i$$

Pour la tâche 0 :

$$T0 : 2-0=2$$

$$T0 : 8-6=2$$

$$T0 : 14-12=2$$

$$T0 : 20-18=2$$

Pour la tâche 1 :

$$T1 : 5-0=5$$

$$T1 : 11-8=3$$

$$T1 : 21-16=5$$

Pour la tache 2 :

$$T2 : 16-0=16$$

4) la laxité nominal

$$L = D - C$$

$$T0 : 6-2=4$$

$$T1 : 8-3=5$$

$$T2 : 24-4=20$$

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

$$\diamond \text{ La gigue de release relative : } RRJ_i = \max_j |(s_{i,j} - r_{i,j}) - (s_{i,j-1} - r_{i,j-1})|$$

• Pour T0 :

$$RRJ_0 = (6 - 6) - (0 - 0) = 0$$

$$RRJ_0 = (12 - 12) - (6 - 6) = 0$$

$$RRJ_0 = (18 - 18) - (12 - 12) = 0$$

$$\text{Donc } RRJ_0 = 0$$

• Pour T1 :

$$RRJ_1 = (8 - 8) - (2 - 0) = 2$$

$$RRJ_1 = (16 - 16) - (8 - 8) = 0$$

$$\text{Donc } RRJ_1 = 2$$

• Pour T2:

$$RRJ_2 = 5 - 0 = 5$$

$$\text{Donc } RRJ_2 = 5$$

❖ La gigue de release absolue : $ARJ_i = \max_j (s_{i,j} - r_{i,j}) - \min_j (s_{i,j} - r_{i,j})$

• Pour T0 :

$$s_{ij} - r_{ij} = 0 - 0 = 0$$

$$s_{ij+1} - r_{ij+1} = 6 - 6 = 0$$

$$s_{ij+2} - r_{ij+2} = 12 - 12 = 0$$

$$s_{ij+3} - r_{ij+3} = 18 - 18 = 0$$

Donc $ARJ_0 = 0$

• Pour T1 :

$$s_{ij} - r_{ij} = 2 - 0 = 2$$

$$s_{ij+1} - r_{ij+1} = 8 - 8 = 0$$

$$s_{ij+2} - r_{ij+2} = 16 - 16 = 0$$

Donc $ARJ_1 = 2 - 0 = 2$

• Pour T2:

$$s_{ij} - r_{ij} = 5 - 0 = 5$$

Donc $ARJ_2 = 5$

❖ La gigue de fin relative : $RFJ_i = \max_j |(f_{i,j} - r_{i,j}) - (f_{i,j-1} - r_{i,j-1})|$

• Pour T0 :

$$RFJ_0 = |(8 - 5) - (2 - 0)| = 1$$

$$RFJ_0 = |(14 - 12) - (8 - 5)| = 1$$

$$RFJ_0 = |(20 - 18) - (14 - 12)| = 1$$

Donc $RFJ_0 = 1$

• Pour T1 :

$$RFJ_1 = |(11 - 8) - (5 - 0)| = 2$$

$$RFJ0 = |(21 - 16) - (11 - 8)| = 2$$

$$\text{Donc } RFJ1 = 2$$

• Pour T2

$$RFJ2 = |16 - 0| = 16$$

$$\text{Donc } RFJ2 = 2$$

❖ La gigue de fin absolue : $AFJ_i = \max_j (f_{i,j} - r_{i,j}) - \min_j (f_{i,j} - r_{i,j})$

• Pour T0 :

$$f_{ij} - r_{ij} = |(11 - 8) - (5 - 0)| = 2$$

$$f_{ij+1} - r_{ij+1} = 8 - 6 = 2$$

$$f_{ij+2} - r_{ij+2} = 14 - 12 = 2$$

$$f_{ij+3} - r_{ij+3} = 20 - 18 = 2$$

$$\text{Donc } AFJ0 = 2 - 2 = 0$$

• Pour T1

$$f_{ij} - r_{ij} = (5 - 0) = 5$$

$$f_{ij+1} - r_{ij+1} = 11 - 8 = 3$$

$$f_{ij+2} - r_{ij+2} = 21 - 16 = 5$$

$$\text{Donc } AFJ1 = 5 - 3 = 2$$

• Pour T2

$$f_{ij} - r_{ij} = |(11 - 8) - (5 - 0)| = 2$$

$$f_{ij+1} - r_{ij+1} = 16 - 0 = 16$$

$$\text{Donc } AFJ2 = 16$$

EXERCICE 2 :

LE CODE :

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

// Fonction exécutée par le thread
void *print_message(void *arg) {
    char *msg = (char *) arg; // Conversion de l'argument vers un pointeur de caractère
    printf("%s\n", msg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t; // Variable pour stocker l'identifiant du thread
    char *msg = "hi there!"; // Message à imprimer

    // Création du thread secondaire
    if (pthread_create(&t, NULL, print_message, msg) != 0) {
        perror("thread creation error");
        return EXIT_FAILURE;
    }

    // Attente de la fin d'exécution du thread secondaire
    if(pthread_join(t, NULL) != 0){
        perror("Error");
        EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

L'exécution de programme :

```
C:\Users\t\Documents\licence d'ex\S6\PTR\github\ex2.exe
hi there!
-----
Process exited after 0.05391 seconds with return value 0
Press any key to continue . . .
```

EXERCICE 3 :

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

Le code :

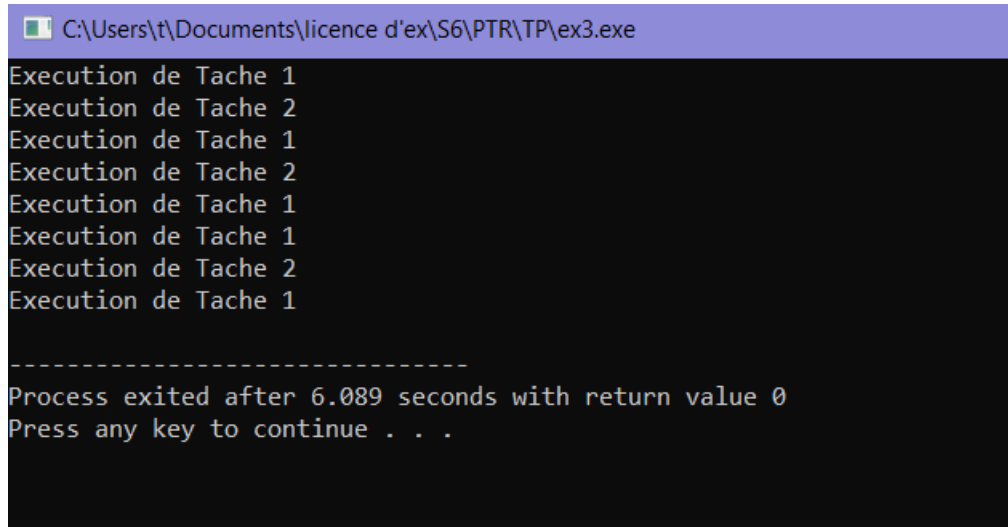
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // N'oubliez pas d'inclure la bibliothèque <unistd.h> pour
utiliser la fonction sleep()

// Fonction exécutée par le premier thread
void *Tache1 (void *arg) {
    int i=0;
    while(i<5){
        printf("Execution de Tache 1\n");
        sleep (1); // Attente de 1 seconde
        i++;
    }
    return NULL;
}
```

```
// Fonction exécutée par le deuxième thread
void *Tache2 (void *arg) {
    int j=0;
    while (j<3){
        printf("Execution de Tache 2\n");
        sleep (2); // Attente de 2 secondes
        j++;
    }
    return NULL;
}

//Test 1
int main(int argc, char *argv[]){
    pthread_t thread1, thread2; // Déclaration des identifiants de thread
    // Création des threads
    pthread_create(&thread1, NULL, Tache1, NULL);
    pthread_create(&thread2, NULL, Tache2, NULL);
    // Attente de la fin des threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return EXIT_SUCCESS;
}
```

L'exécution de programme :



```
C:\Users\t\Documents\licence d'ex\S6\PTR\TP\ex3.exe
Execution de Tache 1
Execution de Tache 2
Execution de Tache 1
Execution de Tache 2
Execution de Tache 1
Execution de Tache 1
Execution de Tache 2
Execution de Tache 1

-----
Process exited after 6.089 seconds with return value 0
Press any key to continue . . .
```


Les deux threads s'exécutent simultanément, c'est-à-dire que les instructions de chaque thread sont entrelacées.

La fonction **pthread_create** permet de créer un nouveau thread et de lui passer une fonction à exécuter.

La fonction **pthread_join** permet d'attendre la fin d'exécution d'un thread.

Dans ce programme, les deux threads s'exécutent en même temps car la fonction **pthread_join** n'est appelée qu'après la création des deux threads.

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

Le code :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // N'oubliez pas d'inclure la bibliothèque <unistd.h> pour
utiliser la fonction sleep()

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

// Fonction exécutée par le deuxième thread
void *Tache2 (void *arg) {
    int j=0;
    while (j<3){
        printf("Execution de Tache 2\n");
        sleep (2); // Attente de 2 secondes
        j++;
    }
    return NULL;
}
```

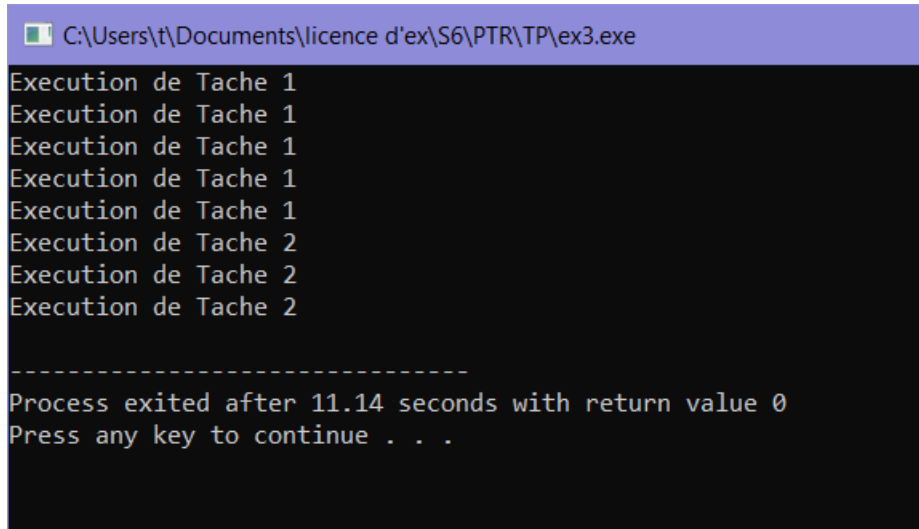
```

}

//Test 2
int main(int argc, char *argv[]){
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, Tache1, NULL);
    pthread_join(thread1, NULL);
    pthread_create(&thread2, NULL, Tache2, NULL);
    pthread_join(thread2, NULL);
    return EXIT_SUCCESS;
}

```

L'exécution de programme :



```

C:\Users\t\Documents\licence d'ex\S6\PTR\TP\ex3.exe
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 1
Execution de Tache 2
Execution de Tache 2
Execution de Tache 2

-----
Process exited after 11.14 seconds with return value 0
Press any key to continue . . .

```

Les deux threads ne s'exécutent pas en même temps.

Dans ce programme, le thread 2 est créé après que le thread 1 ait terminé son exécution.

La fonction **pthread_join(thread1, NULL)** bloque le programme principal jusqu'à ce que le thread 1 ait terminé son exécution.

Une fois que le thread 1 a terminé, le thread 2 est créé et commence à s'exécuter.

3) Expliquer la différence entre les deux résultats.

Dans le premier programme, la fonction **pthread_join** n'est appelée qu'après la création des deux threads. Cela signifie que les deux threads sont créés et commencent à s'exécuter simultanément.

Le système d'exploitation partage le temps processeur entre les deux threads, ce qui entraîne un chevauchement de leurs exécutions.

Dans le deuxième programme, la fonction **pthread_join(thread1, NULL)** est appelée avant la création du thread 2. Cela signifie que le programme principal attend que le thread 1 ait terminé son exécution avant de créer le thread 2. Le thread 2 ne commence donc à s'exécuter qu'après que le thread 1 a terminé.

EXERCICE 4 :

Le code :

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

void *thread_func1(void *arg) {
    printf("Thread 1: Bonjour !\n");
    return NULL;
}

void *thread_func2(void *arg) {
    printf("Thread 2: Salut !\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t1;
    pthread_t t2;

    pthread_create(&t1, NULL, thread_func1, NULL);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, thread_func2, NULL);
    pthread_join(t2, NULL);
    return EXIT_SUCCESS;
}
```

L'exécution de programme :

```
C:\Users\t\Documents\licence d'ex\S6\PTR\TP\ex4.exe
Thread 1: Bonjour !
Thread 2: Salut !

-----
Process exited after 0.06046 seconds with return value 0
Press any key to continue . . .
```

EXERCICE 5 :

Le code :

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

#define NUM_TASKS 3 // Definir le nombre de taches

typedef struct {
    int id;
    int period; // in seconds
} PeriodicTask; // Renommage de la structure pour plus de clarté

void *taskFunction(void *arg) {
    PeriodicTask *task = (PeriodicTask *)arg; // Le pointeur arg est converti en un
    // pointeur PeriodicTask* pour accéder aux membres de la structure (id et periode)

    while (1) {
        sleep(task->period); // The thread pauses execution implementing the
        // task's execution interval
        printf("Task %d executed.\n", task->id); // prints a message indicating the
        // task's ID and execution
```

```
}  
  
    return NULL;  
}  
  
int main() {  
    int TASK_PERIODS[]={1, 2, 3}; // Tableau des périodes d'exécution des tâches  
    // (en secondes)  
    pthread_t threads[NUM_TASKS]; // Tableau pour stocker les identifiants des  
    // threads  
    PeriodicTask tasks[NUM_TASKS]; // Tableau de structures de données des tâches  
    int i;  
  
    // Création des tâches et des threads  
    for (i = 0; i < NUM_TASKS; ++i) {  
        tasks[i].id = i + 1;  
        tasks[i].period = TASK_PERIODS[i];  
        pthread_create(&threads[i], NULL, taskFunction, (void *)&tasks[i]);  
    }  
  
    // Attente de 10 secondes pour laisser les tâches s'exécuter  
    sleep(10);  
  
    // Arrêt et attente de la fin des threads  
    for (i = 0; i < NUM_TASKS; ++i) {  
        pthread_cancel(threads[i]);  
        pthread_join(threads[i], NULL);  
    }  
    return 0;  
}
```

EXERCICE 5 :

Le code :

```
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
  
// Structure pour stocker les informations sur la partie du tableau que chaque  
// thread doit traiter  
typedef struct {
```

```

int
*tab;      // Pointeur vers le tableau
int debut;    // Indice de début de la partie du tableau
int fin;      // Indice de fin de la partie du tableau
int sum_partiel; // Somme partielle calculée par chaque thread
} partiel;

pthread_mutex_t lock; // Déclaration du verrou

// Fonction exécutée par chaque thread pour calculer la somme partielle de sa
partie du tableau
void* sum_partial(void* arg) {
    partiel* var = (partiel *)arg;
    int star = var->debut;
    int end = var->fin;
    int i;
    var->sum_partiel = 0;
    for (i = star; i <= end; i++) {
        pthread_mutex_lock(&lock); // Verrouillage pour éviter les accès
concurrents à sum_partiel
        var->sum_partiel += var->tab[i]; // Calcul de la somme partielle
        pthread_mutex_unlock(&lock);    // Déverrouillage
    }
    pthread_exit(NULL); // Terminaison du thread
}

int main(int argc, char *argv[]) {
    const long taill = 50; // Taille du tableau
    const int nb_thread = 5; // Nombre de threads
    pthread_mutex_init(&lock, NULL); // Initialisation du verrou

    int array[taill];
    int j;
    // Initialisation du tableau
    for (j = 0; j < taill; j++) array[j] = j;

    int x = taill / nb_thread; // Taille de chaque partie du tableau
    pthread_t thread[nb_thread]; // Tableau de threads
    partiel tableau[nb_thread]; // Tableau de structures partielles
    int n;
    // Création des threads
    for (n = 0; n < nb_thread; n++) {
        tableau[n].tab = array; // Initialisation du pointeur vers le tableau
        tableau[n].debut = n * x; // Calcul du début de la partie du tableau
    }
}

```

```
        if (tableau[n].fin == taill) tableau[n].fin = taill - 1; // Vérification
de la fin du tableau
        else tableau[n].fin = tableau[n].debut + x - 1; // Calcul de la fin de la
partie du tableau
        // Création du thread en passant la fonction sum_partial et la structure
comme arguments
        if (pthread_create(&thread[n], NULL, sum_partial, (void *)&tableau[n]) !=
0) {
            perror("Erreur lors de la création du thread");
            return EXIT_FAILURE;
        }
    }
    int w;
    // Attente de la fin de chaque thread
    for (w = 0; w < nb_thread; w++) {
        if (pthread_join(thread[w], NULL) != 0) {
            perror("Erreur lors de l'attente du thread");
            return EXIT_FAILURE;
        }
    }

    int sum = 0;
    int k;
    // Calcul de la somme totale
    for (k = 0; k < nb_thread; k++) {
        sum += tableau[k].sum_partiel;
    }

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

    pthread_mutex_destroy(&lock); // Destruction du verrou

    return 0;
}
```

L'exécution de programme :

```
C:\Users\t\Documents\licence d'ex\S6\PTR\TP\EX5-2.exe
Task 3 executed.
Task 2 executed.
Task 1 executed.
Task 1 executed.
Task 2 executed.
Task 1 executed.
Task 3 executed.
Task 1 executed.
Task 2 executed.
Task 1 executed.
Task 1 executed.
Task 3 executed.
Task 2 executed.
Task 1 executed.
Task 1 executed.
Task 2 executed.
Task 1 executed.
Task 3 executed.
Task 1 executed.
Task 2 executed.
Task 1 executed.
Task 1 executed.
Task 3 executed.
Task 2 executed.
Task 1 executed.
Task 1 executed.
Task 2 executed.
Task 1 executed.
Task 3 executed.
```


L'exécution de programme :

```
C:\Users\t\Documents\licence d'ex\S6\PTR\github\ex6.exe
Somme totale = 1225
-----
Process exited after 0.06869 seconds with return value 0
Press any key to continue . . .
```