




26/04/2024

Compte rendu TP1

Programmation temp réel



Réalisé par : Basma Elrhiraki et Nour
ElHouda Nmer.

Exercice 1

Le facteur d'utilisation d'utilisation du processeur :

Pour T_0 :

$$U_0 = C_0 / P_0 = 2/6 = 1/3 = 0.33$$

Pour T_1 :

$$U_1 = C_1 / P_1 = 3/8 = 0.37$$

Pour T_2 :

$$U_2 = C_2 / P_2 = 4/24 = 1/6 = 0.16$$

Le facteur de charge :

Ici $D = P$

Pour T_0 :

$$Ch_0 = C_0 / D_0 = 2/6 = 1/3 = 0.33$$

Pour T_1 :

$$U_1 = C_1 / D_1 = 3/8 = 0.37$$

Pour T_2 :

$$U_2 = C_2 / D_2 = 4/24 = 1/6 = 0.16$$

Le temps de réponse

Pour T_0 :

$$Tr_0 = f_0 - r_0 = 2 - 0 = 2$$

Pour T_1 :

$$Tr_1 = f_1 - r_1 = 5 - 0 = 5,3,5$$

Pour T_2 :

$$Tr_2 = f_2 - r_2 = 16 - 0 = 16$$

La laxité nominale :

Pour T_0 :

$$L_0 = D_0 - C_0 = 6 - 2 = 4$$

Pour T_1 :

$$L_1 = D_1 - C_1 = 8 - 3 = 5$$

Pour T_2 :

$$L_2 = D_2 - C_2 = 24 - 4 = 20$$

La gigue de release relative :

$$RRJ_0 = \max |(s_{ij+1} - r_{ij+1}) - (s_{ij} - r_{ij})|$$

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

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

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

Donc $RRJ_0 = 0$

Pour T_1 :

$$RRJ_1 = \max |(s_{ij+1} - r_{ij+1}) - (s_{ij} - r_{ij})|$$

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

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

Donc $RRJ_1 = 2$

Pour T_2 :

$$RRJ_2 = \max |(s_{ij+1} - r_{ij+1}) - (s_{ij} - r_{ij})|$$

$$RRJ_2 = 5 - 0 = 5$$

Donc $RRJ_1 = 5$

La gigue de release absolue :

Pour T_0 :

$$ARJ_0 = \max |(s_{ij} - r_{ij}) - \min (s_{ij} - r_{ij})|$$

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

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

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

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

Donc $ARJ_0 = 0$

Pour T_1 :

$$ARJ_1 = \max |(s_{ij} - r_{ij}) - \min (s_{ij} - r_{ij})|$$

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

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

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

Donc $ARJ_1 = 2 - 0 = 2$

Pour T_2 :

$$ARJ_2 = \max |(s_{ij} - r_{ij}) - \min (s_{ij} - r_{ij})|$$

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

Donc $ARJ_2 = 5$

La gigue de fin relative

Pour T_0

$$RFJ_0 = \max |(f_{ij+1} - r_{ij+1}) - (f_{ij} - r_{ij})|$$

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

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

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

Donc $RFJ_0 = 1$

Pour T_1

$$RFJ_1 = \max |(f_{ij+1} - r_{ij+1}) - (f_{ij} - r_{ij})|$$

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

$$RFJ_1 = |(21 - 16) - (11 - 8)| = 2$$

Donc $RFJ_1 = 2$

Pour T_2

$$RFJ_2 = \max |(f_{ij+1} - r_{ij+1}) - (f_{ij} - r_{ij})|$$

$$RFJ_2 = |16 - 0| = 16$$

Donc $RFJ_2 = 2$

La gigue de fin absolue :

Pour T_0

$$AFJ_0 = \max |(f_{ij} - r_{ij})| - \min |(f_{ij} - r_{ij})|$$

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

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

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

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

Donc $AFJ_0 = 2 - 2 = 0$

Pour T_1

$$AFJ_1 = \max |(f_{ij} - r_{ij})| - \min |(f_{ij} - r_{ij})|$$

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

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

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

Donc $AFJ_1 = 5 - 3 = 2$

Pour T_2

$$AFJ_2 = \max |(f_{ij} - r_{ij})| - \min |(f_{ij} - r_{ij})|$$

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

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

$$\text{Donc } AFJ_2 = 16$$

Exercice 2

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

// 1. Définition de la fonction print_message qui affiche un message passé en argument
void *print_message(void *ptr) {
    char *message = (char *)ptr; // Conversion du pointeur en chaîne de caractères
    printf("%s\n", message); // Affichage du message
    pthread_exit(NULL); // Fin du thread
}

int main() {
    pthread_t thread; // Identifiant du thread
    char *message = "Bonjour, je suis un thread !"; // Message à afficher

    // 3. Création d'un thread en lui passant la fonction print_message et le message comme argument
    if (pthread_create(&thread, NULL, print_message, (void *)message)) {
        fprintf(stderr, "Erreur lors de la création du thread\n");
        return 1; }

    // 4. Attente de la fin de l'exécution du thread
    if (pthread_join(thread, NULL)) {
        fprintf(stderr, "Erreur lors de l'attente de la fin du thread\n");
        return 1; }

    return 0; // 6. Fin du programme
}
```

Exercice 3

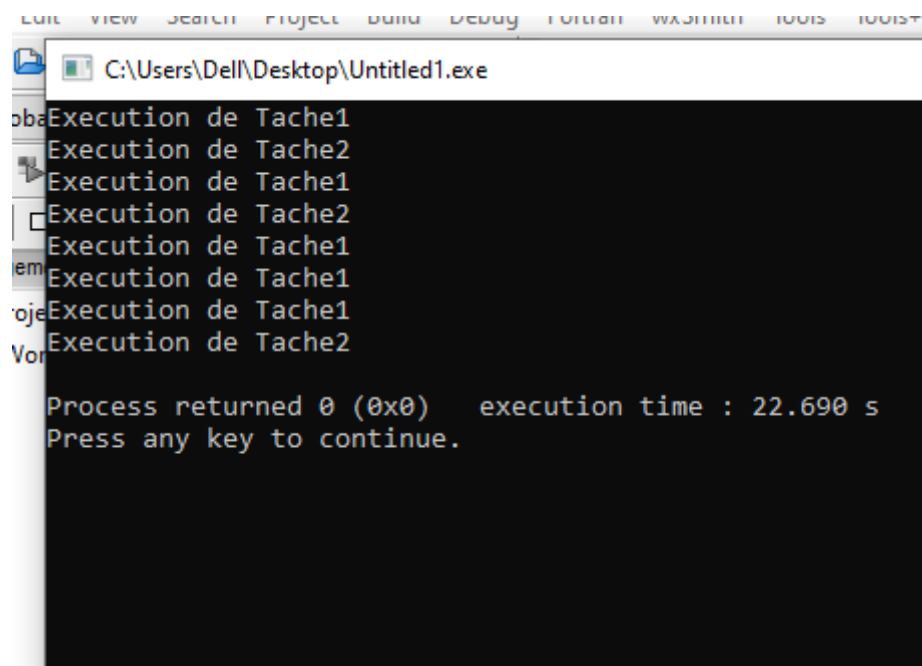
Exemple1:

```
// Fonction exécutée par le premier thread
void *Tache1(void *arg){
    int i = 0;
    while(i < 5) {
        printf("Execution de Tache1\n"); // Affiche un message
        sleep(1); // Pause d'une seconde
        i++; }
    return NULL; // Termine le thread}

// Fonction exécutée par le deuxième thread
void *Tache2(void *arg)
{ int j = 0;
    while(j < 3) {
        printf("Execution de Tache2\n"); // Affiche un message
        sleep(2); // Pause de deux secondes
        j++; }
    return NULL; // Termine le thread}

int main(int argc, char *argv[])
{
    pthread_t thread1, thread2; // Identifiants des threads
    // Création du premier thread
    pthread_create(&thread1, NULL, Tache1, NULL);
    // Création du deuxième thread
    pthread_create(&thread2, NULL, Tache2, NULL);
    // Attente de la fin de l'exécution du premier thread
    pthread_join(thread1, NULL);
    // Attente de la fin de l'exécution du deuxième thread
    pthread_join(thread2, NULL);
    return EXIT_SUCCESS; // Fin du programme avec un code de succès}
```

Après l'exécution du code :



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

Process returned 0 (0x0)   execution time : 22.690 s
Press any key to continue.
```

Exemple 2:

```

// représente la tâche exécutée par thread 1
void *Tache1(void *arg) {
    int i = 0;
    while(i < 5) { // afficher "Execution de Tache1" cinq fois avec un intervalle de 1 seconde entre
chaque affichage.
        printf("Execution de Tache1\n");
        sleep(1);
        i++; }
    return NULL;}

// représente la tâche exécutée par thread 2
void *Tache2(void *arg) {
    int j = 0;
    while(j < 3) { // afficher "Execution de Tache2" trois fois avec un intervalle de 2 secondes entre
chaque affichage.
        printf("Execution de Tache2\n");
        sleep(2);
        j++; }
    return NULL;}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2;

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

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

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

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

    return EXIT_SUCCESS;

}

return EXIT_SUCCESS; // Fin du programme avec un code de succès
}

```


Après l'exécution du code :

```
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 returned 0 (0x0)   execution time : 23.715 s
Press any key to continue.
```

Exercice 4

```

#include <pthread.h> // Inclut la bibliothèque pthread pour gérer les threads
#include <stdio.h>   // Inclut les fonctions d'entrée/sortie standard
#include <stdlib.h>  // Inclut les fonctions standard de gestion de la mémoire
#include <unistd.h>  // Inclut la fonction sleep pour suspendre l'exécution d'un thread

// Fonction exécutée par le premier thread
void *thread_func1(void *arg){
    printf("Thread1: Bonjour!\n"); // Affiche un message
    return NULL; // Termine le thread
}

// Fonction exécutée par le deuxième thread
void *thread_func2(void *arg){
    printf("Thread2: Salut!\n"); // Affiche un message
    return NULL; // Termine le thread
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2; // Identifiants des threads

    // Création du premier thread
    pthread_create(&thread1, NULL, thread_func1, NULL);

    // Création du deuxième thread
    pthread_create(&thread2, NULL, thread_func2, NULL);

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

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

    return EXIT_SUCCESS; // Fin du programme avec un code de succès
}

```

Exercice 5 :

```

typedef struct { // Définit une structure nommée PeriodTask
    int id;      // Un champ pour l'identifiant de la tâche
    int P;      // Un champ pour le temps de période de la tâche
} PeriodTask;

void *TaskFunction(void *arg){ // Définit la fonction TaskFunction qui sera exécutée par chaque thread
    PeriodTask task = (PeriodTask)arg; // Convertit l'argument en un pointeur vers PeriodTask
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL); // Active l'annulation des threads
    while(1){ // Boucle infinie
        sleep(task->P); // Pause l'exécution du thread pendant la période spécifiée dans P
        printf("task %d executed\n", task->id); // Affiche un message indiquant l'exécution de la tâche
        pthread_testcancel(); // Teste si le thread a été annulé }
    }
    return NULL; // Renvoie NULL (cette ligne ne sera jamais atteinte)}

int main(){ // Définit la fonction principale du programme
    int nt = 3; // Déclare et initialise la variable nt à 3 (nombre de tâches)
    int Task_P[] = {1, 2, 3}; // Déclare et initialise un tableau représentant les périodes des tâches
    pthread_t Tr[nt]; // Déclare un tableau de threads
    PeriodTask T[nt]; // Déclare un tableau de structures PeriodTask pour stocker les informations sur les tâches
    for(int i = 0; i < nt; i++){ // Boucle pour chaque tâche
        T[i].id = i + 1; // Affecte un identifiant unique à chaque tâche
        T[i].P = Task_P[i]; // Affecte la période spécifiée à chaque tâche
        pthread_create(&Tr[i], NULL, TaskFunction, (void *)&T[i]); // Crée un thread pour chaque tâche, en passant les informations de la tâche comme argument
        sleep(2); // Délai de 2 secondes entre la création de chaque thread
        for(i = 0; i < nt; i++){ // Boucle pour annuler et attendre chaque thread
            pthread_cancel(Tr[i]); // Annule le thread
            pthread_join(Tr[i], NULL); // Attend la terminaison du thread } }
    return EXIT_SUCCESS; // Termine le programme avec succès}

```

Exercice 6 :

```

#include <stdio.h> //les fonctions d'entrée/sortie standard.
#include <stdlib.h> //les fonctions standard de gestion de la mémoire.
#include <pthread.h> //les bibliothèques nécessaires pour utiliser les threads en C.
#define ARRAY_SIZE 10 // Définit la taille du tableau.
#define NUM_THREADS 4 // Définit le nombre de threads à utiliser.
int totalSum = 0; // Variable globale pour stocker la somme totale des éléments du tableau.
typedef struct { // creation d'une structure.
    int *start; // Pointeur vers le début de la section du tableau à traiter.
    int *end; // Pointeur vers la fin de la section du tableau à traiter.
    pthread_mutex_t *lock; // Pointeur vers un verrou pour synchroniser l'accès à la variable globale
totalSum.
} PartialSumArgs;

void *sum_partial(void *args) { // Fonction exécutée par chaque thread pour calculer la somme partielle
des éléments du tableau.
    PartialSumArgs *partialArgs = (PartialSumArgs *)args;
    int partialSum = 0;
    int *p;
    for (p = partialArgs->start; p < partialArgs->end; p++) {
        partialSum += *p; }
    pthread_mutex_lock(partialArgs->lock); // Verrouille l'accès à la variable globale totalSum.
    totalSum += partialSum; // Ajoute la somme partielle à la somme totale.
    pthread_mutex_unlock(partialArgs->lock); // Déverrouille l'accès à la variable globale totalSum.
    pthread_exit(NULL); // Termine le thread.}

int main() { // Fonction principale du programme.
    int array[ARRAY_SIZE]; // Déclare un tableau d'entiers.
    int i; // Déclare un variable i .
    for (i = 0; i < ARRAY_SIZE; ++i) { // Initialise le tableau avec des valeurs croissantes.
        array[i] = i + 1; }
    pthread_mutex_t lock; // Déclare un verrou pour synchroniser l'accès à la variable globale totalSum.
    pthread_mutex_init(&lock, NULL); // Initialise le verrou.
    pthread_t threads[NUM_THREADS]; // Déclare un tableau de threads.
    PartialSumArgs threadArgs[NUM_THREADS]; // Déclare un tableau d'arguments pour les threads.

```

```

int sectionSize = ARRAY_SIZE / NUM_THREADS; // Calcule la taille de chaque section du
tableau pour chaque thread.

for (i = 0; i < NUM_THREADS; ++i) {

    // Initialise les arguments pour chaque thread.

    threadArgs[i].start = (array + i * sectionSize); // Pointe vers le début de la section du tableau
à traiter.

    // Pointe vers la fin de la section du tableau à traiter.

    threadArgs[i].end = (array + ((i == NUM_THREADS - 1) ? ARRAY_SIZE : (i + 1) * sectionSize));

    threadArgs[i].lock = &lock; // Passe le verrou à chaque thread.

    // Crée chaque thread pour traiter une section du tableau.

    if (pthread_create(&threads[i], NULL, sum_partial, (void *)&threadArgs[i]) != 0) {

        fprintf(stderr, "Erreur lors de la création du thread %d\n", i);

        return 1;    }}

// Attend la fin de l'exécution de chaque thread.

for (i = 0; i < NUM_THREADS; ++i) {

    pthread_join(threads[i], NULL); }

// Affiche la somme totale des éléments du tableau.

printf("Somme totale : %d\n", totalSum);

pthread_mutex_destroy(&lock); // Détruit le verrou.

return 0; // Termine le programme avec succès.

}

```