



COMPTE RENDU TP1 PROGRAMMATION TEMPS REELS

Professeur : Mr Yassin OUKDACH



**REALISE PAR :
SOUMAYA TOMZEN
Bouchra IDBAATMANI**

Exercice 1

1. Le facteur d'utilisation du processeur :

- Pour T₀ :

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

- Pour T₁ :

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

- Pour T₂ :

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

2. Le facteur de charge :

Ici D = P

- Pour T₀ :

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

- Pour T₁ :

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

- Pour T₂ :

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

3. Le temps de réponse

- Pour T₀ :

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

- Pour T₁ :

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

- Pour T₂ :

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

4. La laxité nominale :

- Pour T₀ :

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

- Pour T₁ :

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

- Pour T₂ :

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

5. La gigue de release relative :

- **Pour T0 :**

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

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

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

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

Donc **RRJ₀ = 0**

- **Pour T1 :**

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

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

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

Donc **RRJ₁ = 2**

- **Pour T2:**

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

$$RRJ_2 = 5 - 0 = 5$$

Donc **RRJ₂ = 5**

6. La gigue de release absolue :

- **Pour T0 :**

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

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

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

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

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

Donc **ARJ₀ = 0**

- **Pour T1 :**

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

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

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

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

$$\text{Donc } \mathbf{ARJ_1 = 2 - 0 = 2}$$

- **Pour T2:**

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

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

$$\text{Donc } \mathbf{ARJ_2 = 5}$$

7. La gigue de fin relative

- **Pour T0**

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

$$\mathbf{RFJ_0 = |(8 - 5) - (2 - 0)| = 1}$$

$$\mathbf{RFJ_0 = |(14 - 12) - (8 - 5)| = 1}$$

$$\mathbf{RFJ_0 = |(20 - 18) - (14 - 12)| = 1}$$

$$\text{Donc } \mathbf{RFJ_0 = 1}$$

- **Pour T1**

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

$$\mathbf{RFJ_1 = |(11 - 8) - (5 - 0)| = 2}$$

$$\mathbf{RFJ_0 = |(21 - 16) - (11 - 8)| = 2}$$

$$\text{Donc } \mathbf{RFJ_1 = 2}$$

- **Pour T2**

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

$$\mathbf{RFJ_2 = |16 - 0| = 16}$$

$$\text{Donc } \mathbf{RFJ_2 = 2}$$

8. La gigue de fin absolue :

- **Pour T0**

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

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

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

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

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

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

- **Pour T1**

$$\text{AFJ1} = \max |(f_{ij} - r_{ij})| - \min |(f_{ij} - r_{ij})|$$

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

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

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

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

- **Pour T2**

$$\text{AFJ2} = \max |(f_{ij} - r_{ij})| - \min |(f_{ij} - r_{ij})|$$

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

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

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

Exercice 2

Question : Création d'une fonction **print_message** :

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void *print_message(void *arg){
6      char *msg = (char *)arg;
7      printf(" %s\n",msg);
8      return NULL;
9  }
10
11  int main(){
12      pthread_t thread;
13      char *msg = "bonjour tout le monde";
14      pthread_create(&thread,NULL,print_message,msg);
15      pthread_join(thread,NULL);
16      return EXIT_SUCCESS;
17  }
18
19  
```

Dans la fonction main, nous créons un thread en utilisant **pthread_create**. Nous lui donnons l'adresse de la fonction **print_message** que nous voulons exécuter dans le thread, ainsi que le message que nous voulons imprimer. Ensuite, avec **pthread_join**, nous attendons que le thread se termine avant de quitter le programme.

Exécution :

```
C:\Users\idbaa\OneDrive\Bureau\Projet_C\exercice_2\bin\Debug\exercice_2.exe
bonjour tout le monde

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

Exercice 3 :

Exécution du premier programme :

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void *Tache1(void *arg){
6      int i = 0;
7      while(i<5){
8          printf("Execution de tache 1\n");
9          sleep(1);
10         i++;
11     }
12     return NULL;
13 }
14
15 void *Tache2(void *arg){
16     int j = 0;
17     while(j<3){
18         printf("Execution de tache 2\n");
19         sleep(2);
20         j++;
21     }
22     return NULL;
23 }
24
25
26 int main(int argc, char *argv[])
27 {
28     pthread_t thread1, thread2;
29     pthread_create(&thread1, NULL, Tache1, NULL);
30     pthread_create(&thread2, NULL, Tache2, NULL);
31     pthread_join(thread1, NULL);
32     pthread_join(thread2, NULL);
33     return EXIT_SUCCESS;
34 }
35
```

Dans la fonction main, nous créons deux threads en utilisant **pthread_create**. Chaque thread est associé à l'une des fonctions de tâche, Tache1 et Tache2. Ensuite, nous utilisons **pthread_join** pour attendre que chaque thread termine son exécution avant de quitter le programme.

Exécution :

```
C:\Users\idbaa\OneDrive\Bureau\Projet_C\exemple_1\bin\Debug\exemple_1.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 returned 0 (0x0)   execution time : 6.083 s
Press any key to continue.
```

Dans ce premier exemple, **pthread_join** attend l'exécution des deux threads l'un après l'autre. Chaque appel à **pthread_join** bloque l'exécution du programme principal jusqu'à ce que le thread spécifié se termine. Dans ce programme, les deux threads sont créés séquentiellement, mais ils peuvent s'exécuter en parallèle (simultanément) après leur création.

Exécution du deuxième programme :

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  void *Tache1(void *arg){
6      int i = 0;
7      while(i<5){
8          printf("Execution de tache 1\n");
9          sleep(1);
10         i++;
11     }
12     return NULL;
13 }
14
15 void *Tache2(void *arg){
16     int j = 0;
17     while(j<3){
18         printf("Execution de tache 2\n");
19         sleep(2);
20         j++;
21     }
22     return NULL;
23 }
24
25
26 int main(int argc, char *argv[])
27 {
28     pthread_t thread1, thread2;
29     pthread_create(&thread1, NULL, Tache1, NULL);
30     pthread_join(thread1, NULL);
31     pthread_create(&thread2, NULL, Tache2, NULL);
32     pthread_join(thread2, NULL);
33     return EXIT_SUCCESS;
34 }
35
```

Quand on exécute le programme, le thread principal crée d'abord le premier thread (thread1) qui exécute Tache1. Ensuite, il attend que thread1 ait terminé avant de créer et de joindre le deuxième thread (thread2) qui exécute Tache2. L'attente se fait avec **pthread_join**, ce qui signifie que le thread principal attend que chaque thread individuel termine son travail avant de continuer.

Exécution :

A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\Users\jdbaa\OneDrive\Bureau\Projet_C\exemple_2\bin\Debug\exemple_2.exe'. The main area of the window is black with white text. The text shows the output of a program: 'Execution de tache 1' is printed five times, followed by 'Execution de tache 2' printed three times. At the bottom, it says 'Process returned 0 (0x0) execution time : 11.145 s' and 'Press any key to continue.'

Dans ce programme, nous créons d'abord le thread 1, attendons son exécution, puis créons le thread 2 et attendons son exécution. Cela signifie que chaque tâche est exécutée séquentiellement, l'une après l'autre, plutôt que simultanément. Cela pourrait être utile si la tâche 2 dépendait des résultats de la tâche 1, ou si nous devions nous assurer que la tâche 1 se termine avant de commencer la tâche 2.

Exercice 4

Dans **main**, nous commençons par créer thread1 en utilisant **pthread_create**. Cette fonction prend quatre arguments : l'identifiant du thread, des attributs de thread facultatifs (ici NULL pour les valeurs par défaut), la fonction à exécuter par le thread (thread_func1 dans ce cas), et les arguments à passer à cette fonction (ici NULL car la fonction ne prend aucun argument).

Après avoir créé thread1, nous utilisons **pthread_join** pour attendre que thread1 se termine avant de continuer. Cela signifie que le thread principal (dans **main()**) attend que thread1 termine son exécution avant de passer à l'étape suivante.

Une fois que thread1 est terminé, nous créons thread2 de la même manière, en utilisant **pthread_create** avec thread_func2. Encore une fois, nous utilisons **pthread_join** pour attendre que thread2 se termine avant de quitter le programme.


```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void *thread_func1(void *arg) {
6      printf("Thread 1 : Bonjour !\n");
7      return NULL;
8  }
9  void *thread_func2(void *arg) {
10     printf("Thread 2 : Salut !\n");
11     return NULL;
12 }
13
14 int main() {
15     pthread_t thread1, thread2;
16     pthread_create(&thread1, NULL, thread_func1, NULL);
17     pthread_join(thread1, NULL);
18     pthread_create(&thread2, NULL, thread_func2, NULL);
19     pthread_join(thread2, NULL);
20     return EXIT_SUCCESS;
21 }
22
23

```

Exercice 5

Dans ce programme, chaque tâche est représentée par une structure **PeriodicTask**, qui contient un identifiant (id) et une période (period). L'identifiant permet de distinguer les différentes tâches, tandis que la période indique à quelle fréquence la tâche doit être exécutée.

La fonction **taskFunction** est la fonction exécutée par chaque thread. Elle reçoit en argument un pointeur vers une structure **PeriodicTask** correspondant à la tâche que le thread doit exécuter. À l'intérieur de cette fonction, il y a une boucle infinie (`while(1)`) qui simule l'exécution périodique de la tâche. À chaque itération, le thread dort pendant la période spécifiée dans la structure **PeriodicTask**, puis affiche l'identifiant de la tâche. Entre chaque période, la fonction **pthread_testcancel()** est appelée pour vérifier s'il y a une demande d'annulation du thread à ce point.

Dans la fonction **main()**, nous créons un tableau de tâches `task` et un tableau de threads `threads`. Ensuite, nous créons un thread pour chaque tâche en utilisant **pthread_create()**, en passant à chaque thread l'adresse de la tâche correspondante dans le tableau `task`.

Après avoir laissé les threads s'exécuter pendant un certain temps (dans ce cas, 10 secondes), nous les annulons en utilisant **pthread_cancel()** dans une boucle. Enfin, nous attendons que chaque thread se termine avec **pthread_join()** pour nous assurer qu'ils ont tous terminé leur exécution avant de quitter le programme.

```

5
6 #define NUM_TACHE 3
7
8 typedef struct {
9     int id;
10    int period;
11 } PeriodicTask;
12
13 void *taskFunction(void *arg) {
14     PeriodicTask *task = (PeriodicTask *)arg;
15     int oldstate;
16     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
17     while (1) {
18         sleep(task->period);
19         printf("Tache %d \n", task->id);
20         pthread_testcancel(); // Permettre l'annulation à ce point
21     }
22     return NULL;
23 }
24
25 int main() {
26     PeriodicTask task[NUM_TACHE] = {
27         {1, 1},
28         {2, 2},
29         {3, 3}
30     };
31
32     pthread_t threads[NUM_TACHE];
33
34     // Création des threads pour exécuter les tâches périodiques
35     for (int i = 0; i < NUM_TACHE; i++) {
36         pthread_create(&threads[i], NULL, taskFunction, &task[i]);
37     }
38
39     // Attente de quelques secondes avant d'annuler les threads
40     sleep(10);
41
42     // Annulation des threads
43     for (int i = 0; i < NUM_TACHE; i++) {
44         pthread_cancel(threads[i]);
45     }
46
47     // Attente de la terminaison des threads
48     for (int i = 0; i < NUM_TACHE; i++) {
49         pthread_join(threads[i], NULL);
50     }

```

Exécution :

Cette exécution montre les tâches en cours d'exécution dans un schéma répétitif. Chaque tâche est exécutée une fois avant que la prochaine tâche ne soit exécutée. Cela est dû au fait que les tâches ont des périodes différentes, et que la tâche avec la période la plus courte (Tâche 1) est exécutée plus fréquemment que les tâches avec des périodes plus longues (Tâches 2 et 3)

```

C:\Users\idbaa\OneDrive\Bureau\Projet_C\exercice_5_version_2\bin\Debug\exercice_5_version_
Tache 1
Tache 2
Tache 1
Tache 3
Tache 1
Tache 2
Tache 1
Tache 1
Tache 3
Tache 2
Tache 1
Tache 1
Tache 2
Tache 1
Tache 3
Tache 1
Tache 2
Tache 1
Tache 3
Tache 1
Tache 2
Tache 1
Tache 3

Process returned 0 (0x0)   execution time : 12.134 s
Press any key to continue.

```

Exercice 6 :

```

#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;

typedef struct {
    int *start;
    int *end;
    pthread_mutex_t *lock;
} PartialSumArgs;

```

La variable globale **totalSum** stocke la somme totale des éléments du tableau. La structure **PartialSumArgs** est utilisée pour transmettre les données nécessaires aux threads, y compris les limites de la section du tableau à traiter et un mutex pour la synchronisation. Ces éléments permettent aux threads de calculer des sommes partielles en parallèle tout en évitant les problèmes de concurrence.

```

16 void *sum_partial(void *args) {
17     PartialSumArgs *partialArgs = (PartialSumArgs *)args;
18
19     int partialSum = 0;
20
21     // Calcul de la somme partielle
22     int *p;
23     for (p = partialArgs->start; p < partialArgs->end; p++) {
24         partialSum += *p;
25     }
26
27     // Verrouillage de la section critique
28     pthread_mutex_lock(partialArgs->lock);
29
30     // Ajout de la somme partielle à la somme totale
31     totalSum += partialSum;
32
33     // Déverrouillage de la section critique
34     pthread_mutex_unlock(partialArgs->lock);
35
36     pthread_exit(NULL);
37 }

```

Cette fonction **sum_partial** est exécutée par chaque thread pour calculer la somme partielle des éléments d'une section spécifique du tableau. Elle prend en argument un pointeur vers une structure **PartialSumArgs** qui contient les informations nécessaires pour délimiter la section du tableau à traiter et un mutex pour assurer la synchronisation. À l'intérieur de la fonction, elle itère à travers la section du tableau spécifiée par **partialArgs->start** et **partialArgs->end**, accumulant la somme des éléments dans la variable **partialSum**. Ensuite, elle verrouille la section critique à l'aide du mutex pour mettre à jour la variable globale **totalSum** en ajoutant la somme partielle calculée. Enfin, elle déverrouille la section critique avant de terminer l'exécution du thread.


Dans la fonction main, un tableau est initialisé avec des valeurs de 1 à **ARRAY_SIZE**. Ensuite, un mutex est initialisé pour la synchronisation. Des threads sont créés, chaque thread calculant la somme partielle d'une section du tableau et utilisant un mutex pour éviter les conflits. Enfin, les threads sont attendus, la somme totale est affichée et le mutex est détruit.

```

39 int main() {
40     int array[ARRAY_SIZE];
41     int i;
42
43     // Initialisation du tableau
44     for (i = 0; i < ARRAY_SIZE; ++i) {
45         array[i] = i + 1;
46     }
47
48     // Initialisation du mutex
49     pthread_mutex_t lock;
50     pthread_mutex_init(&lock, NULL);
51
52     // Création des threads et des arguments
53     pthread_t threads[NUM_THREADS];
54     PartialSumArgs threadArgs[NUM_THREADS];
55
56     int sectionSize = ARRAY_SIZE / NUM_THREADS;
57     for (i = 0; i < NUM_THREADS; ++i) {
58         // Configuration des arguments pour chaque thread
59         threadArgs[i].start = (array + i * sectionSize);
60         threadArgs[i].end = (array + ((i == NUM_THREADS - 1) ? ARRAY_SIZE : (i + 1) * sectionSize));
61         threadArgs[i].lock = &lock;
62
63         // Création du thread et vérification des erreurs
64         if (pthread_create(&threads[i], NULL, sum_partial, (void *)&threadArgs[i]) != 0) {
65             fprintf(stderr, "Erreur lors de la création du thread %d\n", i);
66             return 1;
67         }
68     }
69
70     // Attente de la fin de chaque thread
71     for (i = 0; i < NUM_THREADS; ++i) {
72         pthread_join(threads[i], NULL);
73     }
74
75     // Affichage de la somme totale
76     printf("Somme totale : %d\n", totalSum);
77
78     // Destruction du mutex
79     pthread_mutex_destroy(&lock);
80
81     return 0;
82 }
83

```

Exécution :

 C:\Users\idbaa\OneDrive\Bureau\Projet_C\EX6V2\bin\Debug\EX6V2.exe

Somme totale : 55

Process returned 0 (0x0) execution time : 0.026 s
Press any key to continue.